



— MATERIAL DIDÁTICO

# GERENCIAMENTO DE PROJETOS DE SOFTWARE

## RESUMO DA UNIDADE

Esta unidade apresentará a importância e as atividades envolvidas no processo de desenvolvimento de software (PDS) para a produção de um produto de software com qualidade, que são: a especificação dos requisitos, o desenvolvimento, validação e evolução. Verá que os processos devem ser organizados de forma a lidar com as mudanças nos requisitos e projetos de software, como é o caso do modelo de Processo Unificado (UP) que integra boas práticas da engenharia de software para criar processos de softwares ajustáveis. E a importância de se realizar testes no software a fim de validar e verificar as funcionalidades do software que foi implementado conforme a especificação realizada na fase de desenvolvimento. Os softwares podem necessitar passar por manutenção, seja ela para corrigir problemas ou inclusão de melhorias e/ou adaptações, como a inserção de novos requisitos. Existem padrões de melhorias de processos de software, como CMMI e o MPS.BR, que podem ser obtidos por organizações para garantirem qualidade e determinado grau de confiança aos seus clientes.

**Palavras-chave:** Processo de Desenvolvimento do Software. Processo Unificado. Melhoria de Processo. Testes.

## SUMÁRIO

<b>APRESENTAÇÃO DO MÓDULO .....</b>	<b>5</b>
<b>CAPÍTULO 1 – PROCESSO DE DESENVOLVIMENTO DE SOFTWARE .....</b>	<b>7</b>
1.1 O Processo de desenvolvimento de software .....	7
1.2 Atividades do processo de desenvolvimento de software .....	7
1.3 Engenharia de requisitos de software .....	9
1.3.1 Especificação de software.....	10
1.3.2 Projeto e implementação de software .....	14
1.4 Análise de viabilidade do software .....	18
1.5 Especificidades do Software .....	20
1.6 Modelagem do Software .....	21
1.6.1 Visões de um software .....	22
1.6.2 Classificação dos diagramas UML .....	23
1.6.3 Diagramas de Casos de Uso .....	24
1.6.4 Diagramas de Classe .....	27
1.6.5 Ferramentas de modelos de softwares .....	31
1.7 Processo Unificado .....	32
<b>CAPÍTULO 2 – PROJETO DO SOFTWARE .....</b>	<b>38</b>
2.1 Arquitetura do software .....	38
2.1.1 Arquitetura em camadas .....	40
2.1.2 Arquitetura de Repositório.....	44
2.1.3 Arquitetura Cliente-Servidor .....	46
2.1.4 Arquitetura de Duto e Filtro .....	48
2.2 Padrões de Projeto.....	49
2.2.1 Singleton .....	50
2.2.2 Adapter.....	51
2.2.3 Mediator .....	52
2.3 Implementação de Software.....	53
2.3.1 Reuso de Software.....	54
2.3.2 Gerenciamento de Configuração .....	56
2.4 Teste de Software .....	57

2.4.1	Teste Unitário.....	59
2.4.2	Teste de Componente.....	60
2.4.3	Teste de sistema.....	61
2.4.4	Teste de usuário .....	61
2.4.5	Desenvolvimento dirigido a testes.....	62
2.5	Suporte e Manutenção do Software.....	63
2.5.1	Suporte a Software.....	64
2.5.2	Treinamento de Software.....	65
2.5.3	Documentação do Software .....	66
<b>CAPÍTULO 3 – MELHORIA CONTÍNUA DO SOFTWARE .....</b>		<b>67</b>
3.1	Modelo CMMI.....	67
3.1.1	Modelo CMMI por estágios .....	71
3.1.2	Modelo CMMI Contínuo .....	72
3.2	Modelo de processo do Software Brasileiro (MPS.BR).....	74
3.2.1	O Modelo de Referência MPS para Software .....	76
3.3	Comparação entre os modelos CMMI e MPS.BR.....	80
3.4	Outras estruturas de melhoria de processo de software.....	82
<b>REFERÊNCIAS.....</b>		<b>84</b>



## APRESENTAÇÃO DO MÓDULO

A atividade de desenvolver softwares é trabalhosa, complexa e propensa a erros. Em um mercado cada vez mais competitivo, é imprescindível desenvolver softwares com alta qualidade. As tentativas de resolver a complexidade, aumentar a qualidade e reduzir os problemas envolvidos no desenvolvimento do software envolveram a definição de processos de desenvolvimento de software (PDS), que organizam as atividades essenciais para ajudar atingir esses objetivos.

Vários processos e modelos foram propostos para apoiar o desenvolvimento de software. Dentre esses muitos modelos de processos, pode-se destacar o modelo em cascata e o modelo incremental. No entanto, todos eles incluem as quatro atividades essenciais propostas pela engenharia de software, que são: (i) especificação dos requisitos do software, parte em que são definidas as funcionalidades e as restrições em relação ao funcionamento do software; (ii) projeto e implementação de software, em que o software deverá ser implementado de forma a atender as especificações definidas na atividade anterior; (iii) validação de software, que tem a função de verificar e validar se o software atende as demandas do cliente; e (iv) evolução de software, atividade responsável por evoluir o software para atender às necessidades de modificações dos clientes. Existem também outras atividades comuns que dão apoio ao processo de software, como a documentação e o gerenciamento de configuração de software.

Abordaremos a modelagem do software através de uma linguagem unificada, a UML, que auxilia nas atividades do PDS e pode ser usada em atividades de qualquer processo de software. A UML descreve um conjunto de diagramas com múltiplas finalidades, sendo muito utilizada durante o levantamento de engenharia de requisitos para auxiliar na extração dos requisitos e também durante o processo de projeto, em que são utilizados para descrever o software para os desenvolvedores o implementarem. Por fim, esses diagramas são aproveitados para documentar a estrutura e a operação do software.

Posteriormente, é retratado a importância de se escolher uma arquitetura correta. Propriedades como desempenho, proteção e disponibilidade são influenciadas pela arquitetura de software adotada.

Finalmente, são apresentados duas das principais abordagens e técnicas de melhoria de processo de software: CMMI e MPS.BR. Essas abordagens são consideradas em fator importante para as empresas que almejam ser competitivas. Dessa forma, elas mantêm em ciclo contínuo de melhoria que podem melhorar a capacidade de seus processos com o objetivo de alcançar melhores níveis de qualidade, tempo e custo de produção. Os processos de melhoria do software: CMMI e MPS.BR foram criados com base nas normas ISO/IEC 12207 e ISO/IEC 15504 e também trabalham com o desenvolvimento e gerenciamento de requisitos.



## CAPÍTULO 1 – PROCESSO DE DESENVOLVIMENTO DE SOFTWARE

### 1.1 O Processo de desenvolvimento de software

A atividade de desenvolver softwares é trabalhosa. E, Bezerra (2015) reforça essa afirmação através de um estudo realizado pelo Standish Group<sup>1</sup>, o *Chaos Report*, que reproduziu os seguintes dados:

- A porcentagem de projetos que são concluídos dentro do prazo estimado é de 10%.
- Cerca de 25% dos projetos de software são descontinuados antes de ser finalizado.
- 60% dos projetos de software custam acima do esperado.
- O atraso médio nos projetos de software é de aproximadamente 1 ano.

As tentativas de resolver a complexidade e reduzir os problemas envolvidos no desenvolvimento do software envolveram a definição de processos de desenvolvimento de software (PDS), que consiste em todas as atividades necessárias para definir, desenvolver, testar e manter um software (BEZERRA, 2015).

Os problemas mais comuns do processo de desenvolvimento de software estão nos requisitos, pois definir com precisão o que o sistema fará é uma ação bem difícil, conforme será explicado em maior detalhes na Seção 1.3 em que é abordado o tema Engenharia de Requisitos.

### 1.2 Atividades do processo de desenvolvimento de software

Segundo Sommerville (2011, pg. 5), “um processo de software é uma sequência de atividades que leva à produção de um produto de software”. O processo de software é complexo e depende de pessoas para tomar decisões e realizar ponderações. Infelizmente, não há um processo ideal para todos os

---

<sup>1</sup> <https://www.standishgroup.com/>

processos de desenvolvimento de software, é por isso que a maior parte das organizações desenvolvem os próprios processos de software.

Os sistemas críticos requerem requisitos bem definidos e, portanto, um processo de software muito bem estruturado. Já para sistemas que necessitam de alterações constantes nos requisitos é recomendado a utilização de um processo menos formal e mais flexível (SOMMERVILLE, 2011).

Os processos de software são categorizados como tradicionais (dirigidos a planos) ou processos ágeis. Os processos tradicionais consistem naqueles que possuem todas as atividades planejadas com antecedência e o progresso é avaliado por comparação com o planejamento inicial. Como exemplo, pode-se citar: o modelo em Cascata, modelo Espiral. Já nos processo ágeis, o planejamento é gradual e é mais fácil alterar o processo de forma a atender as necessidades de alterações dos clientes (SOMMERVILLE, 2011). O XP (eXtreme Programming) e SCRUM são exemplos de metodologias que adotam o processo ágil.

Sommerville (2011, pg. 19) “em organizações nas quais a diversidade de processos de software é reduzida, os processos de software podem ser melhorados pela padronização”. A padronização é importante para introdução de novos métodos e técnicas de engenharia de software, conforme veremos no Capítulo 3, em que será discutido a melhoria no processo de software.

Existem vários processos de desenvolvimento de software distintos e, de acordo com Sommerville (2011), todos eles incluem quatro atividades fundamentais para a engenharia de software e, conseqüentemente, a engenharia de requisitos. As quatro atividades são:

- Especificação de software: em que são definidas as funcionalidades e as restrições do software em relação ao seu funcionamento.
- Projeto e implementação de software: no qual o software deverá ser desenvolvido de forma que atenda às especificações.
- Validação de software: essa atividade é feita para assegurar que às demandas do cliente foram atendidas.
- Evolução de software: o software deve permitir a evolução de forma que atenda às necessidades de alterações solicitadas pelos clientes.



Essas atividades serão explicadas em mais detalhes nas seções 1.3.1 a 1.3.4. Elas incluem outras subatividades, como validação de requisitos, projeto de arquitetura, testes unitários, dentre outras. Há também outras atividades que dão apoio ao processo de software, como a documentação e o gerenciamento de configuração de software.

### 1.3 Engenharia de requisitos de software

De acordo com Sommerville (2011), os requisitos de software trazem detalhes sobre o que o sistema deve fazer, quais serão os serviços que ele deverá oferecer e quais são as limitações para ele funcionar. Requisitos de software tratam-se do detalhamento dos serviços e as restrições que o software desenvolvido deverá conter. Os requisitos de software são classificados como requisitos funcionais e requisitos não funcionais:

- Requisitos funcionais. Consistem em declarações de serviços que o software deve fornecer e de como será o comportamento do sistema em determinadas situações (SOMMERVILLE, 2011). Por exemplo, em um sistema de biblioteca alguns requisitos funcionais seriam: (i) permitir a consulta ao acervo pelo(s) nome(s) do(s) autor(es); (ii) permitir a consulta ao acervo pelo título do livro; (iii) permitir o cadastro de usuários.
- Requisitos não funcionais. Refere-se as restrições a serviços ou funções oferecidas pelo software, isto é, aborda os requisitos que não estão diretamente relacionados com os serviços oferecidos pelo software a seus usuários, como o tempo de processamento, restrições impostas por normas ou leis, dentre outras (SOMMERVILLE, 2011). Dessa forma, considerando um sistema para bibliotecas, como exemplo de requisitos não funcionais teríamos: (i) limite de tempo para retornar uma pesquisa por autor ou título do livro deve ser no máximo de 3 segundos; (ii) realização de backups dos arquivos do sistema e (iii) armazenar os dados no banco de dados Oracle.

Os requisitos de software podem trazer pelo menos dois problemas para os engenheiros de software, que são: (i) compreender os requisitos, pois, muitas vezes, o cliente tem dificuldade em expressar suas necessidades ou então, nem sabe o que é necessário que o software faça para atender sua demanda e (ii) alteração de requisitos durante o desenvolvimento do software, pois, muitas vezes, o cliente percebe a necessidade de incluir ou mesmo retirar alguma funcionalidade.

Os problemas expostos anteriormente contribuíram para a elaboração de uma abordagem - a engenharia de requisitos - para facilitar a compreensão das necessidades dos clientes antes de se projetar o software. Pois, a engenharia de requisitos fornece um mecanismo para compreender aquilo que o cliente deseja através da análise de suas necessidades, avaliação da viabilidade, negociação de uma solução plausível, especificação da solução sem ambiguidades, validação da especificação e gerenciamento das necessidades à medida que são convertidas em um sistema operacional (PRESSMAN, 2016).

A engenharia de requisitos, conforme visto na Seção 1.2, abrange quatro atividades distintas: especificação de software (Seção 1.3.1), projeto e implementação de software (Seção 1.3.2), validação de software (Seção 1.3.3) e evolução de software (Seção 1.3.4). É importante ressaltar que existem diversos processos de desenvolvimento de software, entretanto essas quatro atividades são comuns a todos eles.

### 1.3.1 Especificação de software

De acordo Sommerville (2011, pg. 24), a especificação de software consiste no “processo de compreensão e definição dos serviços requisitados do sistema e identificação de restrições relativas à operação e ao desenvolvimento do sistema”. Essa atividade é muito importante e, ao mesmo tempo, crítica, pois erros poderão gerar problemas no projeto e na implementação do sistema.

O processo de engenharia de requisitos, que tem como objetivo produzir um documento de requisitos de software, em que especifica um sistema que atenda aos requisitos definidos pelos stakeholders<sup>2</sup>. Os requisitos, normalmente, são

---

<sup>2</sup> Stakeholders são todos aqueles com algum interesse no sistema, ou seja, são as partes interessadas.

apresentados em dois níveis de detalhamento: requisitos de usuários e requisitos de sistemas.

Os requisitos dos usuários tratam-se de declarações abstratas dos requisitos de sistema para o cliente e usuários finais do sistema, portanto precisam de uma declaração de requisitos em alto nível para facilitar a compreensão dos stakeholders. Já os requisitos do sistema tratam-se de uma descrição mais detalhada da funcionalidade a ser fornecida para que os desenvolvedores compreendam as necessidades dos stakeholders. A Figura 1 apresenta um exemplo de um sistema de vendas para exemplificar a distinção entre os requisitos de usuários e de sistemas.

**Figura 1 - Requisitos de Usuário e de Sistemas.**

#### **Definição de Requisitos de Usuários**

- 1- O sistema deve gerar relatórios mensais que mostrem a quantidade de produtos vendidos no mês vigente.

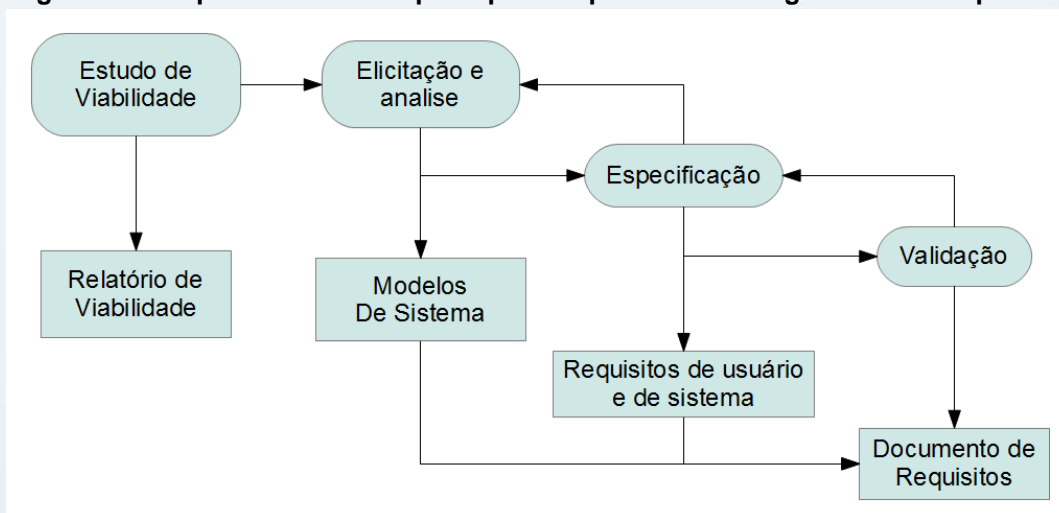
#### **Especificação de Requisitos de Sistemas**

- 1.1- No último dia útil de cada mês deve ser gerado um relatório dos produtos mais vendidos.
- 1.2- O acesso aos relatórios deve ser restrito a usuários autorizados.

**Fonte: Elaborada pela autora, 2020.**

Conforme ilustra a figura 2, o processo de engenharia de requisitos possui quatro principais atividades, sendo elas (SOMMERVILLE, 2011):

**Figura 2 – As quatro atividades principais do processo de engenharia de requisitos**



Fonte: Concurso Interno Progrid<sup>3</sup>. Acesso em: 21/06/2020

1. Estudo de viabilidade. Consiste em um estudo que avalia se o software proposto será rentável a partir de um ponto de vista de negócio. É realizada uma estimativa sobre a possibilidade de atenderem as necessidades do usuário identificado utilizando as tecnologias atuais de software e hardware. O resultado desse estudo deve informar se o projeto do software deve avançar ou não. Mais detalhes serão apresentados na seção 1.4.
2. Elicitação e análise de requisitos. Trata-se da atividade de obtenção de requisitos do sistema por meio do diálogo com possíveis usuários e clientes, realização da análise de tarefas, dentre outras. Esse processo poderá desenvolver um ou mais modelos de sistemas e protótipos, que permitirão aos desenvolvedores compreenderem o sistema a ser especificado.
3. Especificação de requisitos. Consiste na atividade de traduzir as informações coletadas durante a atividade de análise em um documento, que relate um conjunto de requisitos. Nesse documento, poderá conter dois tipos de requisitos: requisitos dos usuários e os requisitos de sistema. A figura 3 ilustra o sumário (títulos e subtítulos esperados) de um pequeno modelo do documento de Especificação de Requisitos de Software. Esse documento contém uma descrição mais detalhada de

<sup>3</sup> <http://progridbb.wikidot.com/engenhariadesoftware>



todas as características do software que se pretende desenvolver. A especificação de requisitos deve ser feita antes do início do projeto do software.

4. Validação de requisitos. Essa atividade é utilizada para verificar os requisitos em relação ao seu realismo, consistência e completude. Durante essa atividade, erros presentes no documento de requisitos são descobertos, portanto deverão ser modificados para corrigir esses problemas.

Essas atividades contidas no processo de requisitos não são feitas em uma única sequência. A atividade de análise de requisitos continua durante a definição e especificação, e novos requisitos surgem durante o processo. Dessa forma, as atividades de análise, definição e especificação de requisitos são intercaladas (SOMMERVILLE, 2011).

Já nos métodos ágeis, a especificação de requisitos não é uma atividade desjunta, mas é vista como parte do desenvolvimento do sistema. Os requisitos são especificados de forma incremental, conforme as prioridades do usuário, e a atividade de elicitação de requisitos é feita pelos usuários que interagem com a equipe de desenvolvimento (SOMMERVILLE, 2011).

Figura 3 - Modelo de especificação de requisitos de software

**Sumário****Histórico de Revisão****1. Introdução**

- 1.1. Propósito
- 1.2. Convenções do documento
- 1.3. Público-alvo e sugestão de leitura
- 1.4. Escopo do projeto
- 1.5. Referências

**2. Descrição geral**

- 2.1. Perspectiva do produto
- 2.2. Características do produto
- 2.3. Classes de usuários e características
- 2.4. Ambiente operacional
- 2.5. Restrições de projeto e implementação
- 2.6. Documentação para usuários
- 2.7. Hipóteses e dependências

**3. Características do sistema**

- 3.1. Características do sistema 1
- 3.2. Características do sistema 2
- 3.3. Características do sistema n

**4. Requisitos de interfaces externas**

- 4.1. Interfaces do usuário
- 4.2. Interfaces de hardware
- 4.3. Interfaces de software
- 4.4. Interfaces de comunicação
- 5. Outros requisitos não funcionais

**5.1. Necessidades de desempenho**

- 5.2. Necessidades de proteção
- 5.3. Necessidades de segurança
- 5.4. Atributos de qualidade de software

**6. Outros requisitos****Apêndice A: Glossário****Apêndice B: Modelos de análise****Apêndice C: Lista de problemas**

Fonte: Adaptado de Pressman, 2016.

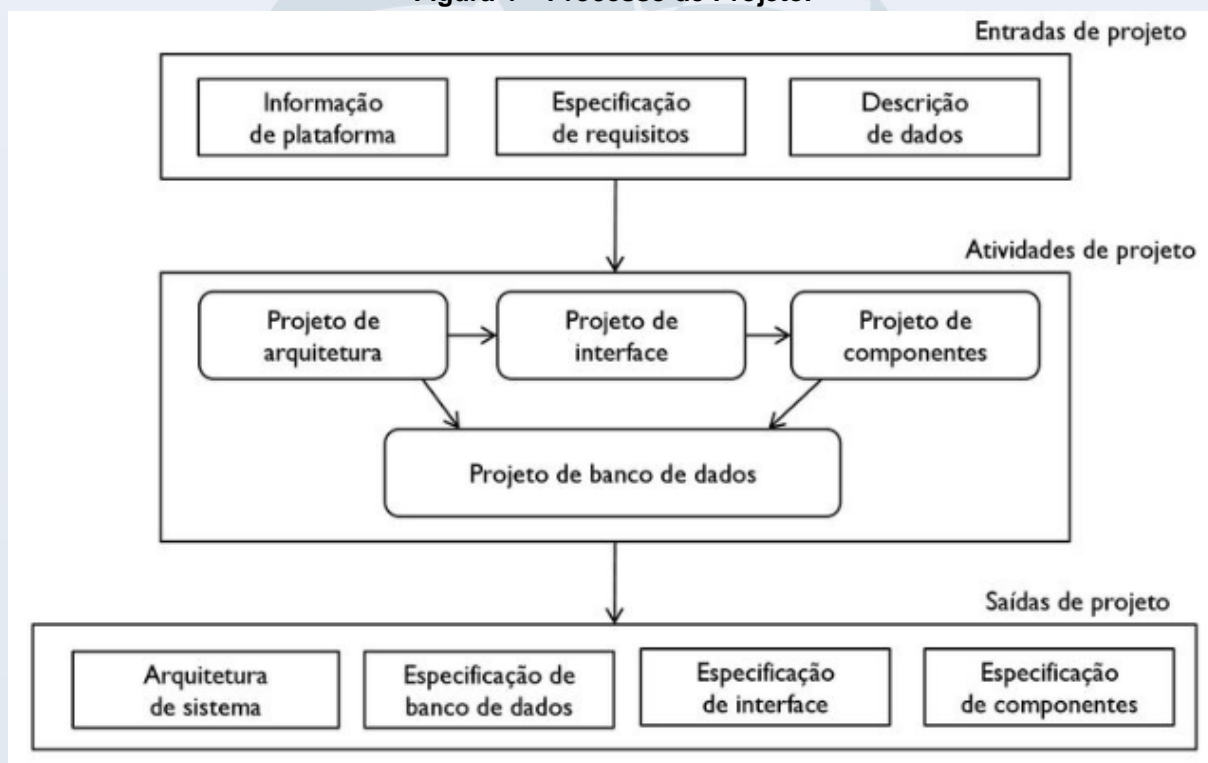
**1.3.2 Projeto e implementação de software**

A atividade de implementação do software trata-se do processo de uma especificação dos requisitos do sistema em um sistema que é executável. Já o

projeto de software consiste em uma descrição da estrutura do software que será implementado, dos modelos e estrutura de dados utilizados pelo sistema, das interfaces entre os componentes do sistema e até dos algoritmos que serão adotados (SOMMERVILLE, 2011).

A figura 4 consiste em um modelo abstrato de processo que mostra as entradas para o processo de projeto, suas atividades e os documentos que são produzidos como saídas do processo de projeto. Como entradas de projeto temos a informação de plataforma (ambiente em que o software será executado), especificação de requisitos (uma descrição da funcionalidade que o software deve oferecer) e descrição de dados (para que a organização dos dados do sistema seja definida).

Figura 4 – Processo de Projeto.



Fonte: Sommerville, 2011.

Já em relação as atividades de projetos temos (SOMMERVILLE, 2011):

- Projeto de arquitetura que permite a identificação da estrutura geral dos sistemas, como os componentes principais, seus relacionamentos e como eles são distribuídos.

- Projeto de interface consiste na definição da interface entre os componentes do sistema.
- Projeto de componente consiste em projetar o funcionamento de cada componente do sistema; e, por último;
- Projeto de banco de dados que consiste em projetar as estruturas de dados do sistema e como devem ser representados no banco de dados. É importante ressaltar que essa atividade não é obrigatória a todos os sistemas, pois pode haver sistemas que não necessitam de banco de dados ou então que podem ser reusados.

A figura 4 leva a interpretação que as atividades do processo de projeto são sequenciais, no entanto elas são intercaladas. Essas quatro atividades de projeto geram um conjunto de saídas do projeto que são: arquitetura de sistema, especificação de banco de dados, especificação de interface e especificação de componentes, conforme se observa na figura 4.

### 1.3.3 Validação de software

A atividade de validação de software é também conhecida por Verificação e Validação (V&V). Essa atividade tem a finalidade de mostrar que um software se adequa a suas especificações ao mesmo tempo que atende as especificações do cliente que encomendou o sistema (SOMMERVILLE, 2011).

A principal técnica de validação é o teste de programas, em que o software é executado com dados de testes simulados. Em suma, avalia se o sistema está adequado aos requisitos. A validação também envolve os processos de verificação, como a inspeção e revisão em cada estágio do processo de software, ou seja, desde a especificação dos requisitos até o desenvolvimento do software (SOMMERVILLE, 2011). Em suma, a verificação tem o objetivo de apurar se o software está de acordo com as especificações estabelecidas no início do projeto.

De acordo com Sommerville (2011), o processo de teste possui três estágios, que são:

1. Testes de componente. Esse tipo de teste é feito pelos desenvolvedores do software. Então, cada componente é testado de forma independente.



Estes podem ser funções ou classes de objetos ou então agrupamentos desses dois. Existem ferramentas de automação de teste, com o JUnit que podem reexecutar testes de componentes quando novas versões dos componentes são implementadas.

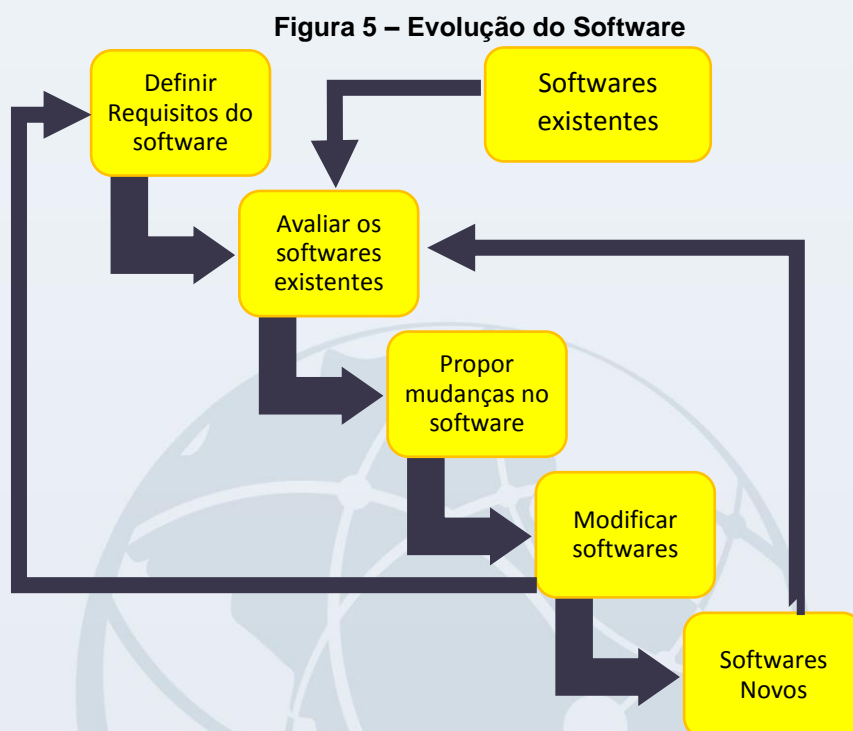
2. Testes de sistema. Nesse tipo de teste, todos os componentes do sistema são integrados para produzir um software completo e, assim, localizar os erros resultantes das interações inesperadas entre os componentes e problemas de interface do componente. Esse teste também possui o objetivo de mostrar que o sistema atende os seus requisitos funcionais e não funcionais.
3. Testes de aceitação. Esse teste é também denominado de Teste do Cliente. Ele consiste no último estágio de teste antes do software ser aceito para uso operacional. O software é testado não mais com dados fictícios e, sim, com os dados reais que foram fornecidos pelo cliente. O que pode revelar erros e omissões na definição dos requisitos do sistema e, também, problemas de requisitos em que os recursos do software não atendem às necessidades do usuário ou desempenho do sistema seja inferior ao aceitável. Em suma, esse tipo de teste mostra quão bem o produto de software atende às necessidades do cliente.

### 1.3.4 Evolução do Software

A evolução do software pode ocorrer a qualquer momento, seja durante ou após o desenvolvimento do software. Segundo Sommerville (2011), historicamente, sempre existiu uma separação entre o processo de desenvolvimento de software (PDS) e o processo de evolução (manutenção) do software.

O PDS é visto pelos programadores como uma atividade criativa, pois requer planejamento, gerenciamento, dentre outros. Já o processo de evolução do software é vista por muitos desenvolvedores como uma atividade monótona e desinteressante comparada a atividade de desenvolver softwares. Entretanto, é cada vez mais irrelevante haver uma distinção entre o PDS e o processo de manutenção do software. Pois, ao invés de pensar nos dois processos separados é mais realista pensar na engenharia de software como um processo evolutivo, pelo fato do

software ser constantemente alterado durante o seu período de vida, conforme exemplifica a figura 5. Essas alterações do software podem estar ligadas às modificações de requisitos e às necessidades do cliente.



Fonte: Adaptada de Sommerville, 2011.

#### 1.4 Análise de viabilidade do software

Análise de viabilidade do software consiste em um tipo de estudo que antecede as etapas de planejamento de um software. Ela é utilizada para ajudar a avaliar a viabilidade do projeto de software, identificar riscos e também os lucros em potencial. Essa etapa é bastante complexa, porém, se os indicadores estiverem corretos, ajudarão o gestor/stakeholders a avaliar e tomar uma decisão mais acertada sobre a aprovação ou não de um projeto de desenvolvimento de software. (GONÇALVES, 2019).

Existem algumas questões, que estão apresentadas na figura 6, que podem ajudar ao gestor/stakeholders a avaliar se o investimento financeiro para desenvolver e distribuir/implantar o software será viável.

**Figura 6 – Questões para ajudar ao gestor/stakeholders avaliar a viabilidade do projeto de software.**

<b>Questão 1</b>	O que é mais viável: criar um novo software ou adotar alguma alternativa disponível no mercado?
<b>Questão 2</b>	O software proposto contribui para a empresa?
<b>Questão 3</b>	Quais são as vantagens do software proposto em relação aos que já existem no mercado?
<b>Questão 4</b>	O que o cliente deseja?
<b>Questão 5</b>	O software proposto resolverá o problema do cliente?
<b>Questão 6</b>	Com base nas condições econômicas da empresa é viável desenvolver o software?
<b>Questão 7</b>	Quais são os riscos do projeto?

**Fonte: Monitoria de Engenharia de Software<sup>4</sup>. Acesso em: 17/06/2020.**

Análise de viabilidade pode ser classificada, como (OLIVEIRA, 2020):

- Viabilidade econômico-financeira tem como objetivo averiguar se o projeto tem a capacidade de proporcionar a recuperação do capital investido e qual será o retorno do investimento, ou seja, análise do custo-benefício.
- Viabilidade operacional consiste em uma análise para verificar a viabilidade operacional. Por exemplo, a empresa ou setor possui recursos, como pessoas, equipamentos, dentre outros em quantidade e qualidade que possibilite que o projeto de software seja executado. Em suma, apurar se a solução está adequada a organização e a que o cliente espera que o software faça.

<sup>4</sup> Disponível em: <https://monitoriadeengenhariadesoftware.wordpress.com/2016/09/06/estudo-de-viabilidade-de-software/>

- Viabilidade técnica refere-se a uma análise que permite verificar se há alguma restrição de equipe, de tecnologias e/ou de recursos técnicos que possibilitem desenvolver softwares em conformidade às especificações. Em resumo, essa viabilidade está vinculada ao suporte técnico que a empresa disponibilizara para o desenvolvimento do sistema.

### SAIBA MAIS

Assunto: Estudo da Viabilidade Econômica de Implantação do Software MRP I Em Uma Micro-Empresa Moveleira Localizada no Vale do Paraíba.

Disponível

em:

<<https://monitoriadeengenhariadesoftware.wordpress.com/2016/09/06/estudo-de-viabilidade-de-software/>>.

## 1.5 Especificidades do Software

De acordo com Sommerville (2011), o quadro 1 descreve as características principais que um software profissional.

**Quadro 1 - Principais características do software profissional.**

<b>Características do software</b>	<b>Descrição</b>
<i>Manutenibilidade</i>	O software deverá ser desenvolvido de maneira que permita a evolução para atender às necessidades dos clientes/stakeholders. A manutenibilidade é considerada um atributo crítico, pois é inevitável não ocorrer mudanças nos softwares.
<i>Confiança e proteção</i>	Um software é considerado confiável quando não causa prejuízos físicos ou econômicos em caso de falha no sistema. Além disso, softwares confiáveis devem conter outras características, como confiabilidade, proteção e segurança.
<i>Eficiência</i>	Um software deverá ter eficiência em relação a sua capacidade de resposta, tempo de processamento,



	uso de memória, dentre outros.
<i>Aceitabilidade</i>	O software é considerado aceitável quando atende as especificidades dos tipos de usuários para o qual ele foi projetado.

Fonte: Sommerville, 2011.

## INDICAÇÕES BIBLIOGRÁFICAS

WAZLAWICK, Raul. **Engenharia de software: conceitos e práticas**. 2 ed. LTC, 2019.

## 1.6 Modelagem do Software

A UML<sup>5</sup> (Linguagem de Modelagem Unificada) é uma linguagem de modelagem visual, ou seja, é um conjunto de notações e semântica correspondente para representar visualmente uma ou mais perspectivas de um sistema. Ela surgiu da união de três técnicas de modelagem: Método de Booch - Grady Booch; Método OOSE - Ivar Jacobson e Método OMT - James Rumbaugh. Os três amigos (Grady Booch, Ivar Jacobson e James Rumbaugh) começaram a unificá-las em meados da década de noventa (BOOCH et al, 2005). Em 1997, a UML foi aprovada como padrão pelo OMG<sup>6</sup> (*Object Management Group*), que é um consórcio internacional de empresas que define e ratifica padrões na área da Orientação a Objetos.

De acordo com Sommerville (2011, pg. 82), a modelagem de software “é o processo de desenvolvimento de modelos abstratos de um sistema, em que cada modelo apresenta uma visão ou perspectiva diferente do sistema”. Em suma, a modelagem é a atividade de construir modelos do sistema. A UML pode ser usada em qualquer processo de software, entretanto é muito utilizada durante o levantamento de engenharia de requisitos para auxiliar na extração dos requisitos e também durante o processo de projeto, em que são utilizados para descrever o software para os desenvolvedores o implementarem. Por fim, esses diagramas são aproveitados para documentar a estrutura e a operação do software (SOMMERVILLE, 2011).

<sup>5</sup> <https://www.uml.org/>

<sup>6</sup> <https://www.omg.org/>

**SAIBA MAIS**

Assunto: Modelagem de Software com a UML

Disponível no link: <https://www.devmedia.com.br/modelagem-de-software-com-uml/20140>

**1.6.1 Visões de um software**

O desenvolvimento de um sistema de software complexo demanda que seus desenvolvedores tenham a possibilidade de examinar e estudar esse sistema a partir de perspectivas diversas. Os autores da UML sugerem que um software pode ser descrito por cinco visões (vide Figura 7) interdependentes desse sistema (BOOCH et al, 2005). Cada visão (perspectivas) enfatiza aspectos variados do sistema, são elas (BOOCH et al, 2005):

**Figura 7 – Visões de um Software**



**Fonte: Elaborado pela autora, 2020.**

- Visão de Caso de Uso: descreve o sistema de um ponto de vista externo, como um conjunto de interações entre o sistema e os agentes externos ao sistema. Esta visão é criada em um estágio inicial e direciona o desenvolvimento das outras visões do sistema.
- Visão de Projeto: destaca as características do sistema que dão suporte, tanto estrutural quanto comportamental, às funcionalidades externamente visíveis do sistema.
- Visão de Implementação: inclui o gerenciamento de versões do sistema, construídas pelo argumento de módulos (componentes) e subsistemas.
- Visão de Implantação: refere à distribuição física do sistema em seus subsistemas e à conexão entre essas partes.
- Visão de Processo: esta visão enfatiza as características de concorrência (paralelismo), sincronização e desempenho do sistema.

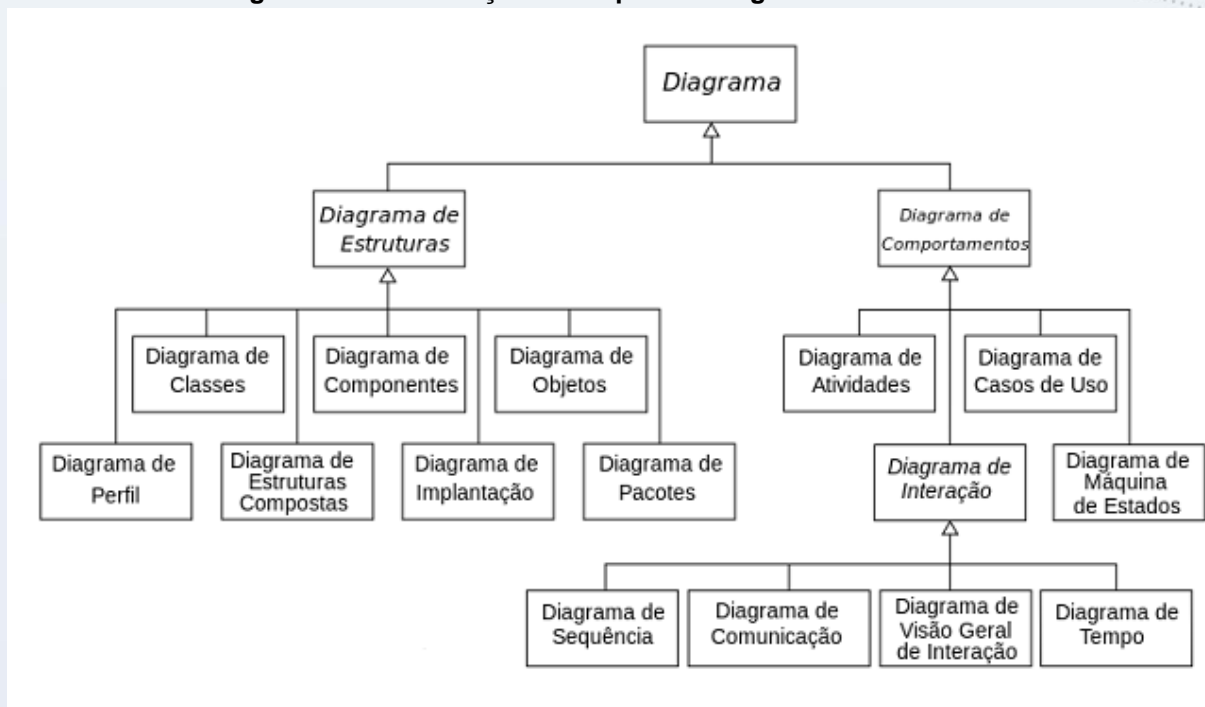
### 1.6.2 Classificação dos diagramas UML

De acordo com Osis and Donins (2017), os diagramas UML (na versão 2.5) são classificados em:

- Diagramas de Estruturais. Esses são destinados a visualizar, especificar, construir e documentar os aspectos estáticos de um software, ou seja, que exibem a estrutura do projeto do software.
- Diagramas Comportamentais. São utilizados para visualizar, especificar, construir e documentar aspectos dinâmicos de um software, ou seja, quando ele está em execução.
- Diagramas de interação. Esses são um subconjunto dos diagramas comportamentais, pois, também, representam os aspectos dinâmicos do software. O diferencial está em mostrar algum tipo de interação que ocorre nos softwares, como interação do usuário através das entradas e saídas de informações, interação entre o software em desenvolvimento com outros ou, então, interação entre os componentes do software (SOMMERVILLE, 2011).

A figura 8 apresenta todos os tipos de diagramas UML da versão 2.5. Nas próximas seções, serão apresentados os diagrama UML mais comuns.

**Figura 8 – Classificação dos Tipos de Diagramas de UML 2.5.**



Fonte: VENTURA, 2019.

### ATENÇÃO

Ninguém consegue usar todos os diagramas UML. Desse modo, os engenheiros de software/desenvolvedores são livres para escolherem um ou um conjunto de diagramas UML e implementar na fase do processo de desenvolvimento de software que optarem.

### 1.6.3 Diagramas de Casos de Uso

O diagrama de casos de uso mostra um conjunto de casos de uso, atores e seus relacionamentos; é usado para organizar e modelar os aspectos dinâmicos de um sistema (OSIS and DONINS, 2017, Pg.40, tradução nossa)<sup>7</sup>.

Os elementos do diagrama de casos de uso são:

<sup>7</sup> Use case diagram shows a set of use cases and actors and their relationships; it is used to organize and model the dynamic aspects – required usages – of a system.



Caso de uso. Descreve o que um sistema ou subsistema faz, mas não especifica como está fazendo. Geralmente, a especificação do Caso de Uso é escrita em linguagem natural, estruturando a descrição como etapas sequenciais executadas pelos atores e pelo sujeito envolvido no cenário do Caso de Uso. O cenário de Caso de Uso inclui o cenário principal e um cenário alternativo que é usado no caso de exceção ou condições específicas se tornarem verdadeiras durante a execução. O Caso de Uso é desenhado como uma elipse mostrando o nome do Caso de Uso dentro dele (OSIS and DONINS, 2017, Pg.40 e 41, tradução nossa)<sup>8</sup>. Caso de Uso é exemplificado na figura <sup>9</sup>.

Ator. Representa um conjunto coerente de usuários, funções, sistemas externos que interagem com o sujeito por meio de Casos de Uso específicos (OSIS and DONINS, 2017, Pg.40, tradução nossa)<sup>9</sup>. A figura 1.9 demonstra como é a representação de um ator.

Relacionamento. Existem três tipos de relacionamentos usados no diagrama de Caso de Uso.

- O primeiro é entre Casos de Uso, mostrando relações de *extensão* e *inclusão* de dependência (OSIS and DONINS, 2017, Pg.41, tradução nossa)<sup>10</sup>.
  - A dependência de *extensão* (*extend*) é usada para mostrar um fluxo alternativo de eventos no caso de condições específicas serem atendidas, por exemplo, um cenário alternativo.
  - A dependência de *inclusão* (*include*) é usada para especificar um cenário comum incluído em vários casos de uso (evita as duplicações dos mesmos cenários/requisitos).

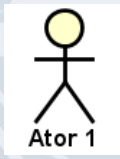

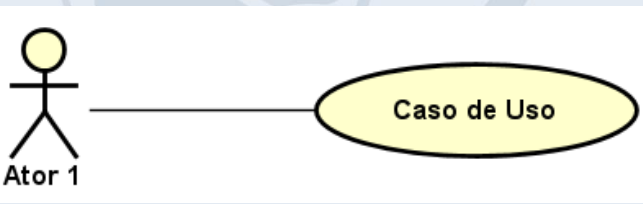
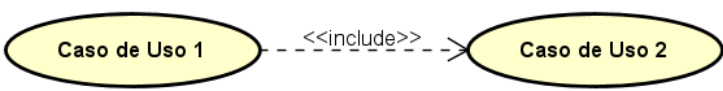
<sup>8</sup> Use case – it describes what a system or subsystem is doing, it does not include and does not specify how it is doing it. Commonly specification of Use Case is written in natural language, structuring the description as sequential steps performed by the actors and the subject involved in the Use Case scenario. Use Case scenario includes both the main scenario and an alternative scenario which is used in the case of exception or specific conditions becoming true during the execution of it. The Use Case is drawn as an ellipsis showing the name of Use Case.

<sup>9</sup> Actor – represents a coherent set of users, roles, external systems that interacts with the subject through specific Use Cases. Actors model entities that are outside the system.

<sup>10</sup> The first one is between Use Cases showing dependency extend and include relations. The extend dependency is used to show an alternate flow of events in the case if specific conditions are met, e.g. an alternate scenario, while the include dependency is used to specify a common scenario included in multiple Use Cases (it avoids the duplications of the same scenarios / requirements).

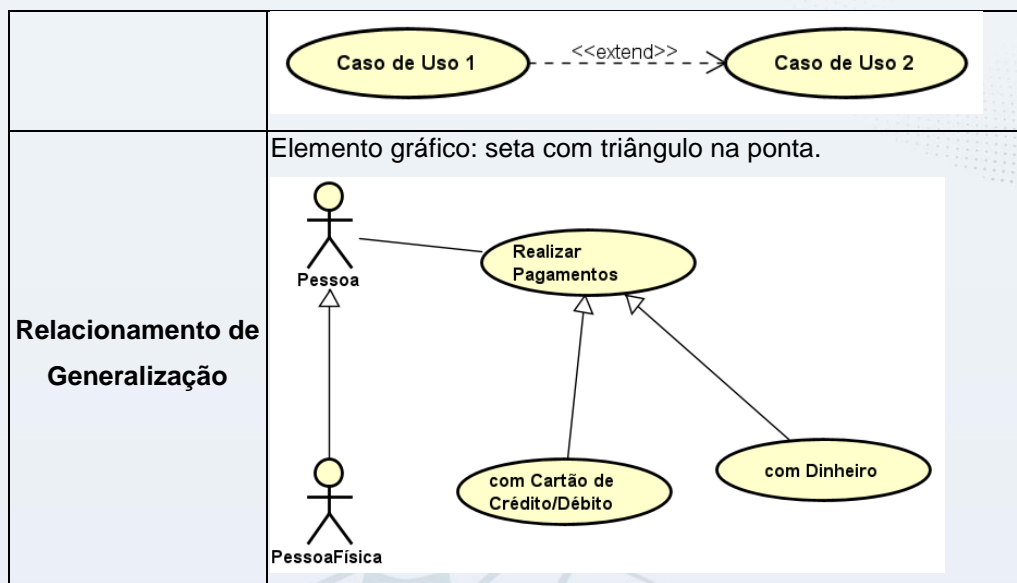
- O segundo tipo de relacionamento é entre Casos de Uso e Atores, normalmente, representados como Associações, mostrando o link de comunicação entre Ator e Caso de Uso (OSIS and DONINS, 2017, Pg.41, tradução nossa)<sup>11</sup>, conforme ilustra a figura 9.
- O último tipo é entre atores – geralmente, a Generalização é usada para mostrar os relacionamentos pai-filho (Herança) entre as papéis (OSIS and DONINS, 2017, Pg.41, tradução nossa)<sup>12</sup>. A generalização ocorre entre dois ou mais casos de uso/atores que possuem características semelhantes.

**Figura 9 – Elementos gráficos do diagrama Casos de Uso.**

Nome	Representação
<b>Ator</b>	Elemento gráfico: boneco palito 
<b>Caso de Uso</b>	Elemento gráfico: elipse 
<b>Relacionamento de Comunicação (Associação)</b>	Elemento gráfico: linha simples 
<b>Relacionamento de Inclusão</b>	Elemento gráfico: seta tracejada com estereótipo: <<include>> 
<b>Relacionamento de Extensão</b>	Elemento gráfico: seta tracejada com estereótipo: <<extend>>

<sup>11</sup> The second type of relationships is between Use Cases and Actors, typically represented as Associations showing the communication link between Actor and Use Case.

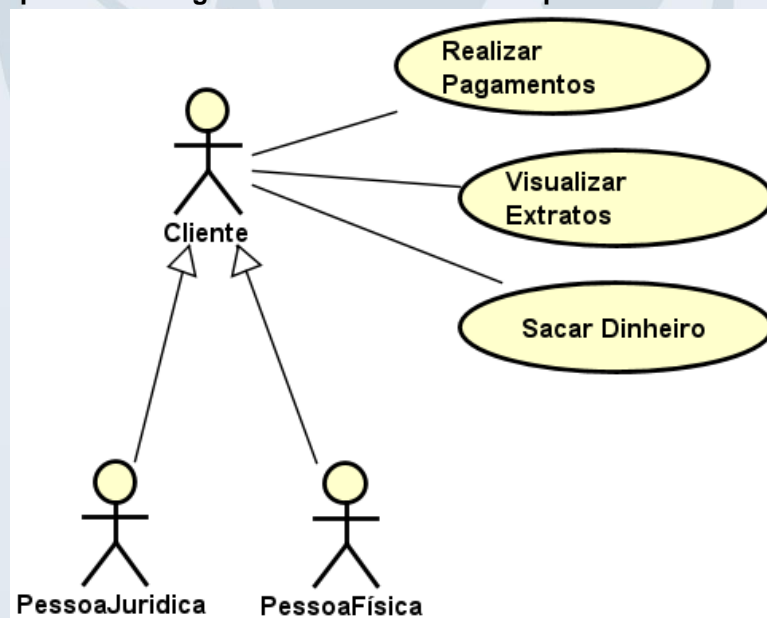
<sup>12</sup> The last type is between actors – typically Generalization is used to show the parent-child relationships between roles.



Fonte: Elaborado pela autora, 2020.

A figura 10 ilustra um diagrama de casos de uso sobre uma movimentação bancária. Nesse diagrama, é possível visualizar diferentes tipos de relacionamentos, como: generalização e inclusão.

Figura 10 – Exemplo de um diagrama Casos de Uso sobre parte de sistema de conta corrente.



Fonte: Elaborado pela autora, 2020.

#### 1.6.4 Diagramas de Classe

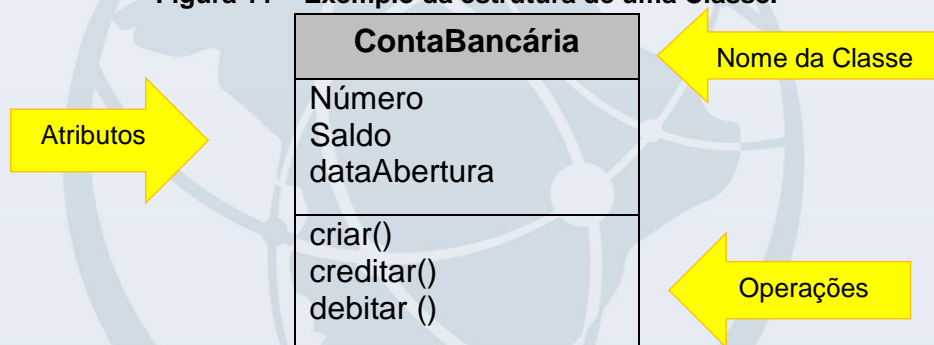
O diagrama de classes é o diagrama mais utilizado para modelagem de sistemas orientados a objetos e é usado para ilustrar o ponto de vista estático de um

sistema. Ele mostra um conjunto de classes, interfaces e seus relacionamentos (OSIS and DONINS, 2017, Pg.22, tradução nossa)<sup>13</sup>.

O diagrama de classe inclui os seguintes elementos:

- **Classes.** Consiste em um modelo para criar objetos, fornecendo especificação de atributos e operações que uma instância da classe pode concluir (OSIS and DONINS, 2017, Pg.22, tradução nossa)<sup>14</sup>. Uma classe é representada por um retângulo dividido em três partes, conforme ilustra a figura 1.11. Observe que o nome da classe aparece na parte superior, nesse exemplo, o nome dessa classe é ContaBancária. Os atributos aparecem na divisão do meio, que são: número, saldo e dataAbertura. As operações (métodos) associados à classe aparecem na última parte da divisão. Nesse exemplo, as operações, são: criar(), creditar() e debitar().

**Figura 11 – Exemplo da estrutura de uma Classe.**



Fonte: Elaborado pela autora, 2020.

- **Relacionamento.** Trata-se de um conceito que especifica algum tipo de relacionamento entre elementos, ou seja, faz referência a um ou mais elementos relacionados. Existem diversos relacionamentos, dentre eles, os mais importantes são (OSIS and DONINS, 2017, Pg.24, tradução nossa)<sup>15</sup>:

<sup>13</sup> Class diagram is the most common diagram found in object-oriented systems and it is used to illustrate the static viewpoint of a system. It shows a set of classes, interfaces, and their relationships.

<sup>14</sup> Class – a template for creating objects, providing specification of attributes and operations that an instance of the class can complete.

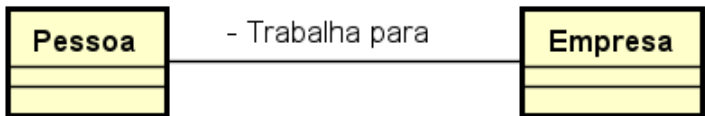
<sup>15</sup> relationship – a concept that specifies some kind of relationship between elements, i.e. it references one or more related elements. The relationship can model either physical or logical relations. Generalization – relates generalized classes to specialized classes, i.e. it shows the parent-child relations or the superclass-subclass relations. Association – structural relationships among classes showing the physical structure of things. Dependency – states that one entity uses the information and services of another entity.

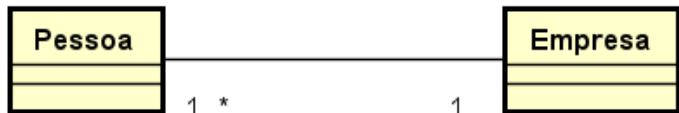
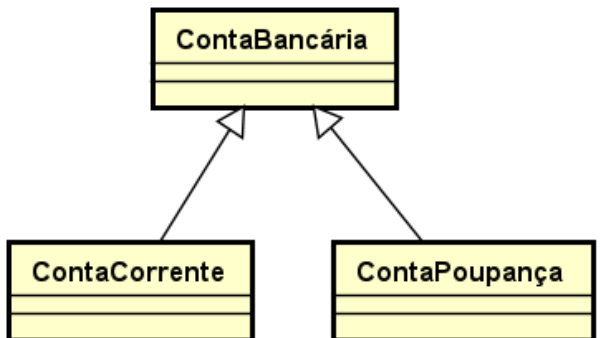
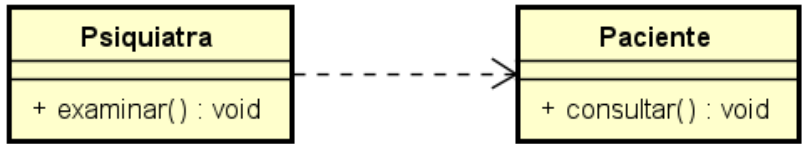

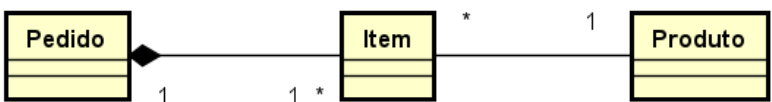


- **Associação.** São as relações estruturais entre classes, mostrando o vínculo entre duas classes. A figura 12 elucida esse tipo de relacionamento.
- **Generalização.** Relaciona classes generalizadas a classes especializadas, ou seja, mostra as relações pai-filho (herança) ou relações de superclasse e subclasse. A figura 12 ilustra esse tipo de relacionamento.
- **Dependência.** Afirma que uma entidade usa as informações e serviços de outra entidade. A figura 12 exemplifica esse tipo de relacionamento.
- **Agregação (Associação Todo-Parte).** Trata-se de um tipo especial de associação. Demonstra que as informações de um objeto precisam ser complementadas por um objeto de outra classe, ou seja, que um objeto está contido em outro. A figura 12 ilustra esse tipo de relacionamento.
- **Composição.** Trata-se de uma variação do tipo agregação, que representa um vínculo mais forte entre objetos-todo e objetos-parte. Em suma, pode-se dizer que existe uma dependência de existência. A figura 12 apresenta esse tipo de relacionamento.

A figura 12 apresenta os elementos do diagrama de Classes e, posteriormente, a figura 13 exemplifica um diagrama de classes para um sistema de compras.

**Figura 12 – Elementos gráficos do diagrama de Classe.**

Nome	Representação
<b>Associação</b>	<p>Elemento gráfico: linha reta que pode ter um nome ou não.</p> 
<b>Multiplicidade</b>	<p>Elemento gráfico:</p> <ul style="list-style-type: none"> <li>▪ Apenas um → 1</li> <li>▪ Zero ou Muitos → 0..*</li> <li>▪ Um ou Muitos → 1..*</li> <li>▪ Zero ou um → 0..1</li> </ul>

	
Generalização	<p>Elemento gráfico:</p> 
Dependência	<p>Elemento gráfico:</p>  <p>Psiquiatra é dependente da classe Paciente.</p>
Agregação	<p>Elemento gráfico:</p>  <p>Um time formado por pessoas, ou seja, pessoas estão contidas em um time.</p>
Composição	<p>Elemento gráfico:</p>  <p>Um pedido inclui vários itens. Cada item diz respeito a um produto.</p>

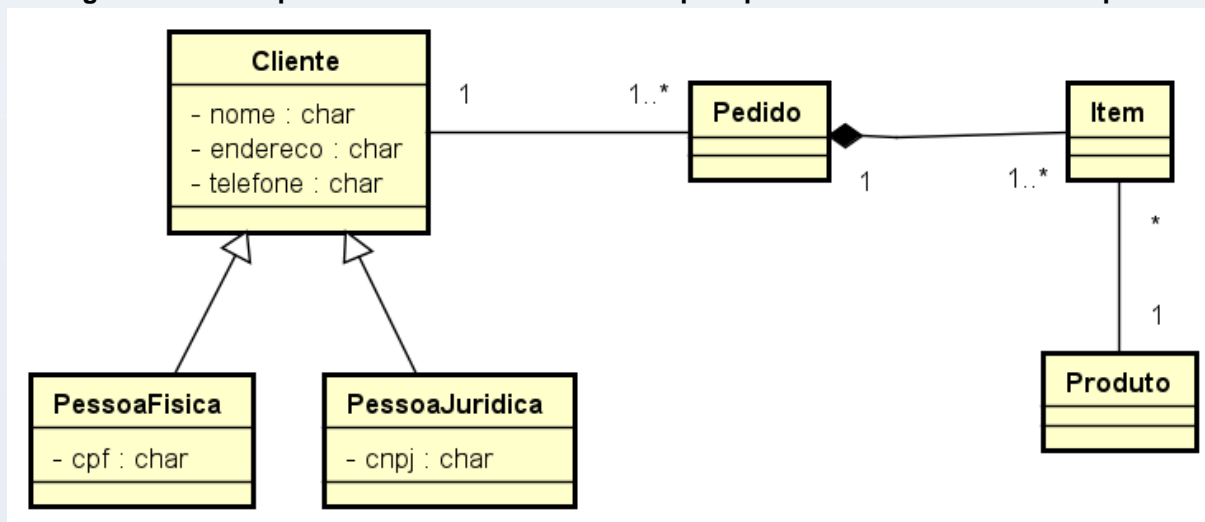
Fonte: Elaborado pela autora, 2020.

**SAIBA MAIS**

Assunto: Diagramas UML

Disponível no link: <<http://josemalcher.net/guia-de-estudos-engenharia-de-software/linguagem-de-modelagem-unificada-uml/paradigma-orientado-objeto/>>.

**Figura 13 – Exemplo da estrutura de uma Classe para parte de um sistema de compras.**



Fonte: Elaborado pela autora, 2020.

**INDICAÇÕES BIBLIOGRÁFICAS**

BEZERRA, Eduardo. **Princípios de análise e projeto de sistemas com UML**. 3 ed. Rio de Janeiro: Campus, 2015.

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. **UML – guia do usuário**. 2 ed. Rio de Janeiro: Elsevier, 2005.

FOWLER, M. **UML Essencial**. Um breve guia para a linguagem-padrão de modelagem de objetos. 3. ed. Porto Alegre, RS: Artmed, 2005.

**1.6.5 Ferramentas de modelos de softwares**

De acordo com Bezerra (2015), o processo de desenvolvimento do software é muito difícil, entretanto essa atividade pode ser descomplicada pelo uso de ferramentas que ajudam na construção de modelos de software, na integração do trabalho de cada pessoa da equipe, dentre outros.

Existem softwares que são usados para auxiliar o ciclo de vida de desenvolvimento de um sistema, como (BEZERRA, 2015):

- as ferramentas CASE (Computer Aided Software Engineering, Engenharia de Software Auxiliada por Computador) consistem em ferramentas que auxiliam diversas atividades na engenharia de software, como: (i) a criação de diagramas UML que possibilitam produzir a perspectiva gráfica do software e (ii) rastreamento de requisitos que permite encontrar os artefatos de software gerados como consequência da existência daquele requisito.
- os ambientes de desenvolvimento integrado (Integrated Development Environment, IDE) que permitem a codificação do software e proporcionar algumas facilidades, como: (i) depuração de código-fonte; (ii) verificação de erro em tempo de execução e (iii) refatoração do código.
- e as ferramentas de suporte ao desenvolvimento, como: (i) a geração de relatório relatando quais partes de um programa não foram testadas; (ii) gerenciamento de versões dos artefatos de software que são implementadas durante o ciclo de vida de um software e (iii) ferramentas que possibilitam ao gerente de projeto possibilidade de gerenciar: o desenvolvimento de cronogramas de tarefas, acompanhamento do progresso das atividades e dos custos.

#### SAIBA MAIS

Assunto: Ferramentas UML

Disponível no link: <<https://www.profissionaisti.com.br/2018/08/7-ferramentas-online-gratuitas-para-criar-diagramas-uml/>>

### 1.7 Processo Unificado

O Processo Unificado (PU ou UP, *Unified Process*) foi criado em 1995 por Jim Rumbaugh, Grady Booch e Ivar Jacobson (DIAS, 2019). Eles desenvolveram o Processo Unificado para ser uma metodologia para engenharia de software Orientada a Objetos (OO) utilizando a UML. O PU trata-se de uma tentativa de



aproveitar as melhores características e recursos dos modelos de processo de software tradicionais, mas, também, incluindo algumas características do desenvolvimento ágil de softwares, como a importância da comunicação com o cliente (PRESSMAN, 2011).

#### REFERÊNCIAS BIBLIOGRÁFICAS

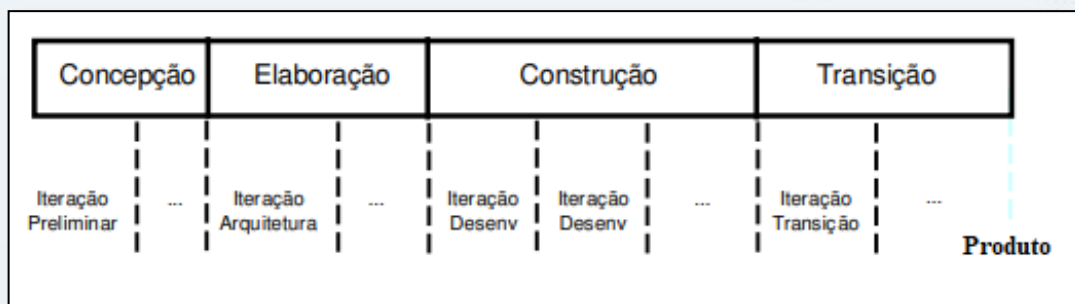
JACOBSON, Ivar; BOOCH, Grady and RUMBAUGH, James. The Unified Software Development Process. Addison-Wesley Professional. 1 ed. 1999.

Para Nakagawa (2012), as principais características do PU, são:

- Desenvolvimento iterativo. O desenvolvimento do software é composto por diversos ciclos de iteração. Em cada ciclo acontece as atividades de análise de requisitos do software, projeto, implementação e teste. Em cada ciclo de iteração, produz um software testado, integrado e executável.
- Orientado a caso de uso. O PU é orientado a caso de usos pelo fato de o utilizar para guiar todo o trabalho de desenvolvimento do software, que vai desde a elicitação dos requisitos até os testes.
- Centrado na arquitetura. A arquitetura consiste na organização do software como um todo, incluindo elementos estáticos, dinâmicos e seus relacionamentos. Dessa forma, ela ajudará a compreender a visão global do software, facilitar o reuso e a evolução do software.

De acordo com Osis e Donins (2017) e Nakagawa (2012), as iterações estão divididas por quatro fases, em que cada fase consiste em uma ou mais iterações, conforme exemplifica a figura 14.

**Figura 14- Fases do ciclo de vida do Processo Unificado.**



**Fonte: Adaptada de Vergilio. Acesso em: 22/06/2020.**

O ciclo de vida do PU baseia-se em diversas repetições ao longo da vida do software, quando ocorre um ciclo completo, tem-se de uma versão do produto do software, em que cada ciclo é composto por quatro fases, que são:

- **Concepção (Inception).** Essa é a primeira e mais curta fase do projeto. É utilizada para: (i) estabelecer a viabilidade de implantar o software, (ii) estimar os custos e cronograma do sistema, (iii) identificação dos potenciais riscos que devem ser acompanhados ao longo do projeto do software, (iv) definição do escopo e do esboço da arquitetura do sistema.
- **Elaboração.** Nessa fase, é possível ter uma visão detalhada do software através da definição dos requisitos sistema (por exemplo, na forma de diagrama de casos de uso), da arquitetura criada na fase anterior (fase de concepção) e no acompanhamento contínuo dos riscos do projeto de software para eliminar ou amenizar o impacto no cronograma final.
- **Construção.** É nessa fase que o software é efetivamente implementado, portanto é a maior fase e mais longa do Processo Unificado. “Durante essa fase, o projeto do software é finalizado e refinado e, assim, o software é construído por meio da base criada durante a fase de elaboração. A fase de construção está dividida em várias iterações, assim, cada iteração resulta em uma versão executável do sistema. A iteração final dessa fase deverá liberar um software completo que será implantado durante a fase de transição” (OSIS E DONINS, 2017, s.p, tradução nossa)<sup>16</sup> (próxima fase).

<sup>16</sup> *Texto Original: During this phase, the design of the system is finalized and refined and the system is built using the basis created during elaboration phase. The construction phase is divided into multiple iterations, for*

- Transição. Essa é a última fase do projeto, em que é entregue o software aos usuários finais/clientes. “Essa fase também inclui a migração de dados de sistemas legados e o treinamento dos usuários” (OSIS E DONINS, 2017, s.p, tradução nossa)<sup>17</sup>.

“Cada fase e sua iteração consistem em um conjunto de atividades predefinidas, conforme ilustrado na figura 15. O Processo Unificado descreve atividades de trabalho como disciplinas - uma disciplina é um conjunto de atividades e artefatos relacionados em uma área de assunto (por exemplo, as atividades na análise de requisitos). As disciplinas descritas pelo Processo Unificado são as seguintes” (OSIS E DONINS, 2017, s.p, tradução nossa)<sup>18</sup>:

- “Modelagem de negócios. Consiste na modelagem de objetos de domínio e modelagem dinâmica dos processos de negócios.
- Requisitos. Refere-se a análise de requisitos do sistema em consideração. Incluindo atividades, como escrever casos de uso e identificar requisitos não funcionais.
- Análise e design. Abrange os aspectos do projeto, como a arquitetura geral.
- Implementação. Trata-se da programação e construção do sistema.
- Teste. Envolve atividades de teste, como planejamento de teste, desenvolvimento de cenários de teste, teste alfa e beta, teste de regressão e teste de aceitação.
- Implantação. Refere-se as atividades de implantação do software desenvolvido” (OSIS E DONINS, 2017, s.p, tradução nossa) <sup>19</sup>.

---

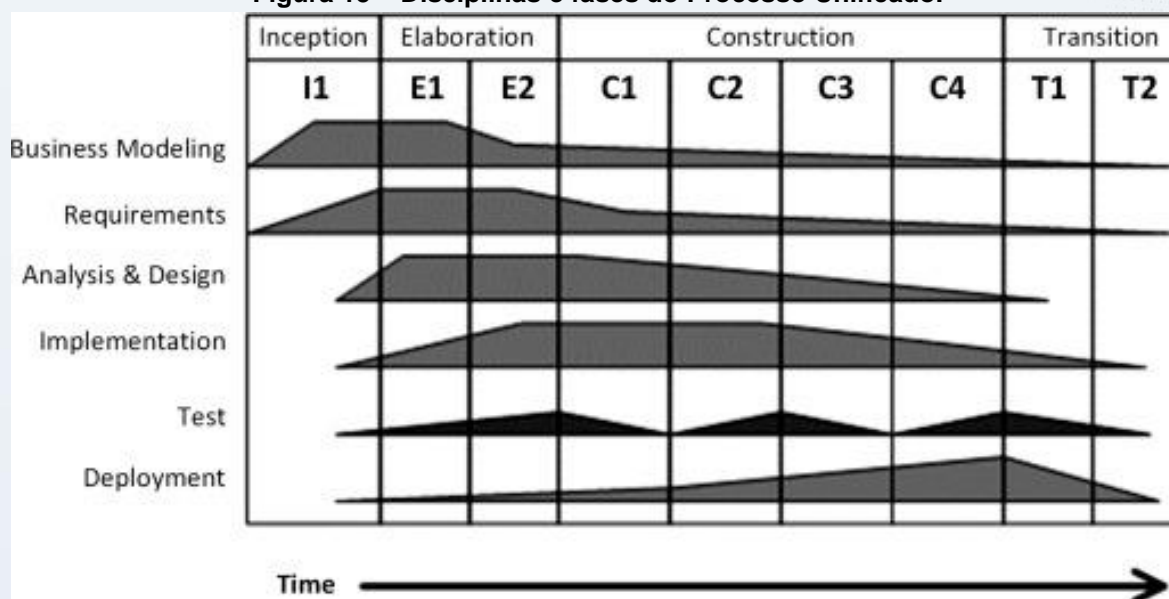
*each iteration to result in an executable release of the system. The final iteration of construction phase releases fully completed system which is to be deployed during transition phase.*

<sup>17</sup> *Transition phase includes also data migration from legacy systems and user trainings.*

<sup>18</sup> *Each phase and its iteration consists of a set of predefined activities. The Unified Process describes work activities as disciplines—a discipline is a set of activities and related artifacts in one subject area (e.g., the activities within requirements analysis). The disciplines described by Unified Process are as follows:*

<sup>19</sup> *Business modeling - domain object modeling and dynamic modeling of the business processes, Requirements - requirements analysis of system under consideration. Includes activities like writing use cases and identifying nonfunctional requirements, Analysis and design - covers aspects of design, including the overall architecture, Implementation - programming and building the system (except the deployment), Test - involves testing activities such as test planning, development of test scenarios, alpha and beta testing, regression testing, acceptance testing, and Deployment - the deployment activities of developed system.*

**Figura 15 – Disciplinas e fases do Processo Unificado.**



Fonte: Traduzida de Osis e Donins (2017).

Existem diversas extensões e adaptações do Processo Unificado (PU), dentre elas pode-se citar:

- Rational Unified Process (RUP). Trata-se de um processo de engenharia de software, que fornece uma abordagem disciplinada para atribuir tarefas e responsabilidades dentro de uma organização de desenvolvimento. Seu objetivo é garantir a produção de alta qualidade software que atenda às necessidades de seus usuários finais, dentro de um cronograma e orçamento previsíveis.
- Agile Unified Process (AUP). Consiste em uma versão simplificada do RUP, pois descreve uma abordagem simples e fácil de entender para o desenvolvimento de software de aplicativos de negócios usando técnicas e conceitos e técnicas ágeis, como o desenvolvimento orientado a teste (TDD).

#### SAIBA MAIS

Assunto: Documentação oficial do RUP

Disponível

no

link:

[https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251\\_bestpractices\\_TP026B.pdf](https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf)



### **SAIBA MAIS**

Assunto: Processo Unificado e suas Extensões

Disponível no link: <http://www.bawiki.com/wiki/Unified-Process.html>



## CAPÍTULO 2 – PROJETO DO SOFTWARE

### 2.1 Arquitetura do software

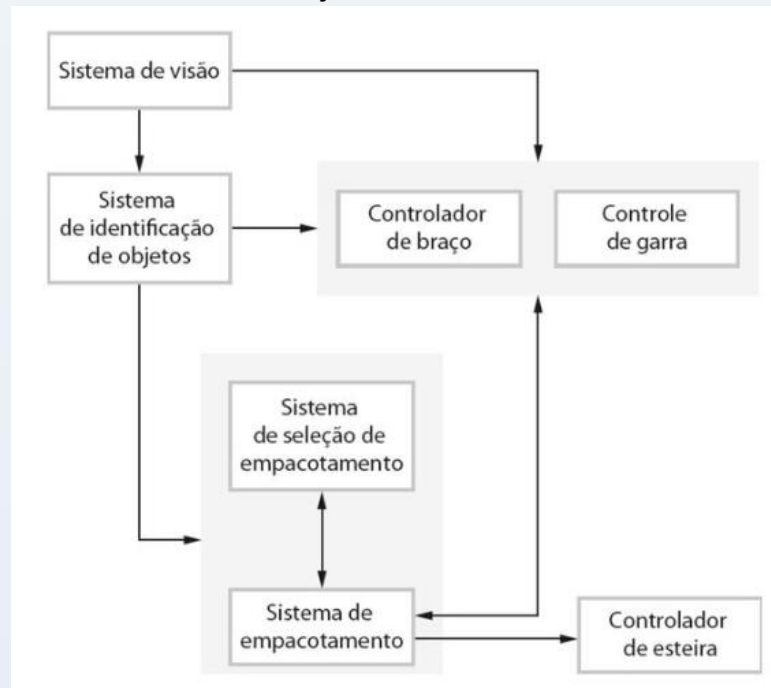
O projeto de arquitetura serve para ilustrar como será a estrutura geral do sistema que será desenvolvido e, conseqüentemente, facilitar a compreensão de como o software deverá estar organizado. De acordo com Sommerville (2011), o projeto de arquitetura é um elo crítico entre o projeto de software e a engenharia de requisitos, porque identifica os principais componentes estruturais de um sistema e os relacionamentos entre eles. Em suma, o projeto de arquitetura do software consiste em um modelo de arquitetura que apresenta como o sistema está organizado em um conjunto de componentes de comunicação.

Para facilitar a compreensão de uma arquitetura do software, considere a figura 16. Essa figura apresenta um modelo abstrato da arquitetura de um sistema de controle robotizado de empacotamento de diferentes tipos de objetos. Nesse modelo são ilustrados os componentes que necessitam ser desenvolvidos e os relacionamentos entre eles. Observe que o sistema a ser desenvolvido deverá ter um componente de visão para selecionar os objetos na esteira, identificar o tipo de objeto e, posteriormente, selecionar o tipo correto de empacotamento para esse objeto. Após empacotar os objetos eles deverão ser colocados em outra esteira (SOMMERVILLE, 2011).

Normalmente, as arquiteturas do software são modeladas através de diagramas de blocos, conforme demonstra a figura 16. Cada caixa representa um componente e as caixas dentro de outras caixas apontam que o componente foi decomposto em subcomponentes. Já as setas indicam que os dados e/ou sinais de controle são passados de um componente a outro na direção das setas (SOMMERVILLE, 2011).

Os diagramas de blocos consistem em uma imagem de alto nível da estrutura do software para facilitar a compreensão de todas as pessoas, como clientes e stakeholders, envolvidas no desenvolvimento do software; eles também são utilizados para descrever a arquitetura do software (SOMMERVILLE, 2011).

**Figura 16 – Projeto de arquitetura de um sistema de controle robotizado de empacotamento de objetos distintos.**



Fonte: Sommerville, 2011.

Segundo Sommerville (2011), as arquiteturas de software podem ser classificadas em dois níveis de abstração:

- Arquitetura em pequena escala. Esse tipo de arquitetura refere-se aos programas individuais, ou seja, preocupa-se com a maneira que um programa individual é decomposto em componentes.
- Arquitetura em grande escala. Já esse tipo de arquitetura trata da arquitetura de sistemas corporativos complexos, que incluem outros sistemas, programas e componentes de programas. Esses sistemas complexos (por exemplo os empresariais, como ERP<sup>20</sup>) estão distribuídos por diversos computadores que podem pertencer e ser administrados por diferentes empresas.

No decorrer do projeto de arquitetura do software, os arquitetos de software, baseando-se em seus conhecimentos e experiências; eles devem tomar diversas decisões estruturais sobre o sistema e seu processo de desenvolvimento. De acordo

<sup>20</sup> ERP é a sigla para *Enterprise Resource Planning*, que é um software integrado de gestão empresarial que engloba em um único software as informações gerenciais dos setores de uma empresa, tais como: Contabilidade, Finanças, Fiscal, Vendas e Recursos Humanos.

com Sommerville (2011), os arquitetos de software devem considerar diversas questões fundamentais sobre o sistema, as quais são apresentadas no quadro 2.

**Quadro 2 – Questões fundamentais para auxiliar os arquitetos de software**

	<b>Descrição das questões fundamentais</b>
<i>Questão 1</i>	Existe uma arquitetura genérica de aplicação que pode atuar como um modelo para o sistema que está sendo projetado?
<i>Questão 2</i>	Como o sistema será distribuído por meio de um número de núcleos ou processadores?
<i>Questão 3</i>	Que padrões ou estilos de arquitetura podem ser usados?
<i>Questão 4</i>	Qual será a abordagem fundamental para se estruturar o sistema?
<i>Questão 5</i>	Como os componentes estruturais do sistema serão decompostos em subcomponentes?
<i>Questão 6</i>	Que estratégia será usada para controlar o funcionamento dos componentes do sistema?
<i>Questão 7</i>	Qual a melhor organização de arquitetura para satisfazer os requisitos não funcionais do sistema?
<i>Questão 8</i>	Como o projeto de arquitetura será avaliado?
<i>Questão 9</i>	Como a arquitetura do sistema deve ser documentada?

**Fonte: Sommerville, 2011.**

A arquitetura de um software poderá basear-se em um determinado padrão de arquitetura, que consiste na descrição de uma organização do software, como uma organização cliente-servidor ou uma arquitetura em camadas. Antes de tomar decisões sobre qual arquitetura de software utilizar, é recomendado que saiba como os diferentes tipos de padrões de arquitetura podem ser usados e quais são os seus pontos fracos e fortes (SOMMERVILLE, 2011). Nas próximas seções (2.1.1 até 2.1.4), será apresentado uma série de padrões, que é, frequentemente, utilizado em diferentes tipos de sistemas.

### **2.1.1 Arquitetura em camadas**

Arquitetura em camadas propõe a criação de aplicativos modulares, em que a camada mais alta se comunica com a camada mais baixa e assim sucessivamente.



Dessa forma, faz com que uma camada seja dependente apenas da camada imediatamente abaixo. A figura 17 exemplifica o funcionamento das três camadas presentes nessa arquitetura, que são (BACALÁ JÚNIOR)<sup>21</sup>. Acesso em: 22/06/2020:

- Camada de Apresentação: Trata-se do código responsável pela apresentação, controle da página e tela de navegação.
- Camada de Negócios: Código relativo à implementação de regras de negócio ou requisitos do sistema.
- Camada de Persistência: Essa camada é a responsável pelo armazenamento e recuperação dos dados quando solicitado. O propósito é garantir uma independência da fonte de dados (arquivos, bancos de dados, etc) e, ao mesmo tempo, manter as informações entre diferentes sessões de uso.

**Figura 17 – Funcionamento das Três Camadas**



**Fonte: Elaborada pela autora, 2020.**

A figura 17, além das três camadas (Apresentação, Negócios e Persistência), também são apresentados: (i) o banco de dados que é a atual representação

<sup>21</sup> BACALÁ JÚNIOR, Sílvio. **Arquitetura em Camadas**. 22 Slides. Disponível em: <<http://www.facom.ufu.br/~bacala/PI/WebCamadas.pdf>>. Acesso em: 22/06/2020.

persistente do estado do sistema; e (ii) as classes assistentes e classes de utilidade que são classes essenciais para o funcionamento ou mesmo o complemento de uma aplicação ou parte dela, por exemplo, o *Exception* para tratamento de erros (BACALÁ JÚNIOR)<sup>21</sup>. Acesso em: 22/06/2020:

A arquitetura em camadas é responsável por organizar o sistema em camadas conforme a funcionalidade relacionada a cada tipo de camada. Uma camada fornece serviços à camada acima dela e as camadas que estão nos níveis mais baixos representam os principais serviços suscetíveis de serem usados em todo o sistema. A arquitetura em camadas é utilizada na construção de novos recursos em cima de sistemas existentes e também quando o sistema está espalhado por várias equipes, em que cada equipe tem a responsabilidade em uma camada de funcionalidade (SOMMERVILLE, 2011).

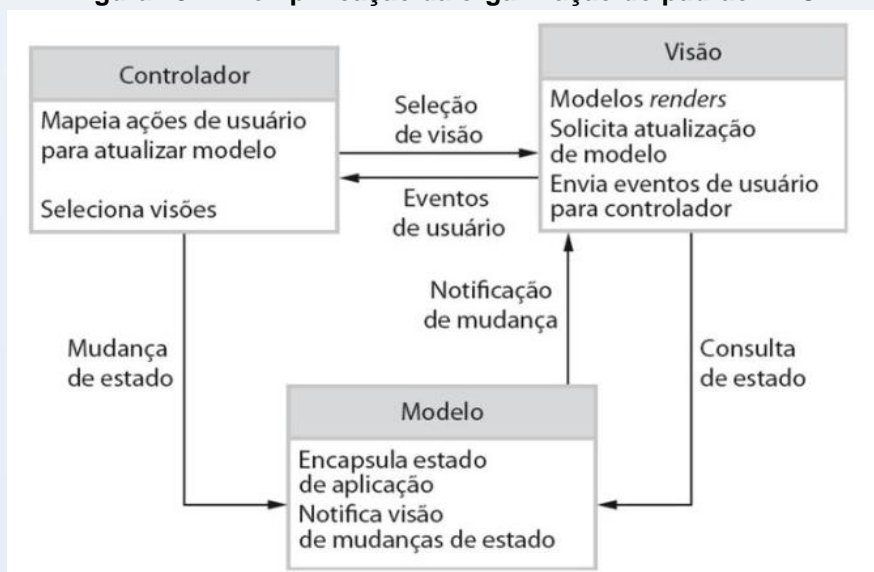
As principais vantagens da arquitetura em camadas são: (i) desde que a interface seja mantida poderá ser feita a substituição de camadas inteiras e (ii) Recursos redundantes como, autenticação pode ser fornecidos em cada camada para aumentar a confiança do sistema. Como desvantagem pode-se citar: (i) a dificuldade em proporcionar uma clara separação entre as camadas e uma camada de alto nível se dá pelo fato de interagir diretamente com as camadas de baixo nível, ao invés da camada imediatamente abaixo dela; (ii) O desempenho poderá ser um problema por causa dos múltiplos níveis de interpretação de uma solicitação de serviço, pois são processados em cada camada (SOMMERVILLE, 2011).

Além do padrão de arquitetura em camada, tem-se o padrão MVC (*Model-View-Controller* ou Modelo-Visão-Controlador). Esse padrão é considerado a base do gerenciamento de interação em muitos sistemas baseados em Web. A função desse padrão é separar a apresentação e a interação dos dados do sistema, que são estruturados em três componentes lógicos (Modelo, Visão e Controle) que interagem entre si (SOMMERVILLE, 2011).

O componente Modelo é responsável por gerenciar o sistema de dados e as operações associadas a esses dados. Ou seja, todas as regras pertinentes com a obtenção, tratamento e validação dos dados devem ser implementados nessa camada. Já o modelo Visão tem a função de definir e gerenciar como os dados serão apresentados ao usuário, por exemplo: página Web, formulário, relatórios,

dentre outros. E, por último, o componente Controlador, essa é a primeira camada a ser executada quando o usuário faz uma requisição ao servidor. Pois, é responsável por gerenciar a interação do usuário, como os cliques do mouse e informar essas interações para os outros dois componentes (Visão e Modelo), (SOMMERVILLE, 2011), conforme ilustra a figura 18.

**Figura 18 – Exemplificação da organização do padrão MVC.**

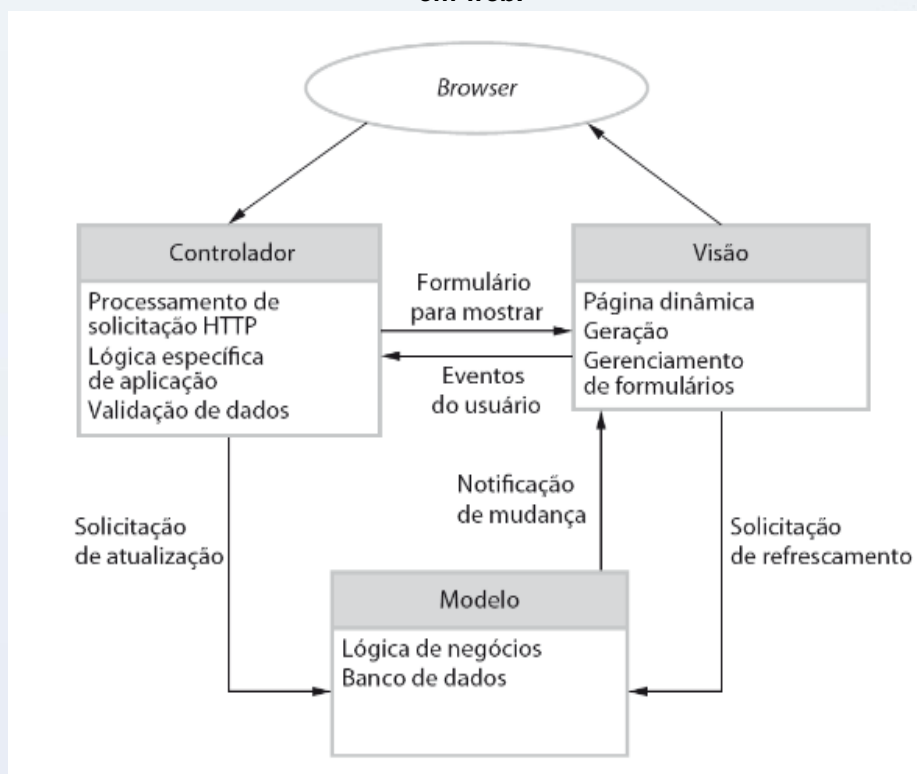


Fonte: Sommerville, 2011.

O padrão MVC é utilizado quando existem várias formas de se visualizar e interagir com dados e, também, quando são desconhecidos os futuros requisitos de interação e apresentação de dados. As principais vantagens desse padrão são: (i) permitir que os dados sejam modificados independente de sua representação e vice-versa; (ii) auxiliar a apresentação dos mesmos dados de formas distintas. Como desvantagem, tem-se: necessidade de um tempo maior para analisar e modelar o sistema. (SOMMERVILLE, 2011).

A figura 19 exemplifica uma possível arquitetura de aplicações Web utilizando o padrão MVC.

**Figura 19 – Padrão MVC utilizado para gerenciamento de interações em um sistema baseado em web.**



Fonte: Sommerville, 2011.

### INDICAÇÕES BIBLIOGRÁFICAS

GAMMA, E.; HELM, R.; JOHNSON, R; VLISSIDES, J. *Design Patterns: elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison-Wesley, 1995.

COPLIN, J. H.; HARRISON, N. B. *Organizational Patterns of Agile Software Development*. Englewood Cliffs, NJ: Prentice Hall, 2004.

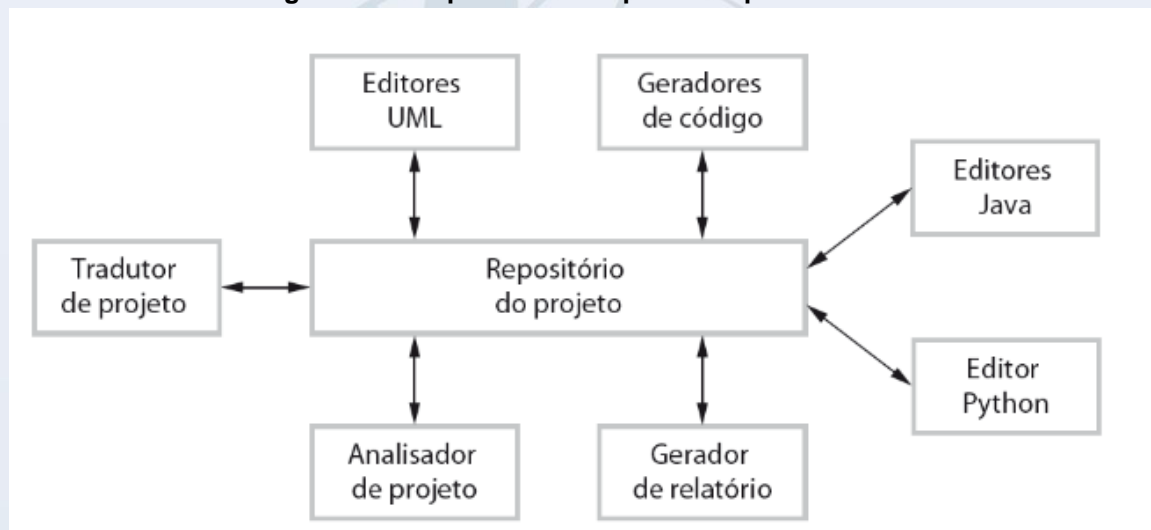
### 2.1.2 Arquitetura de Repositório

A maioria dos softwares que utilizam grandes quantidades de dados está organizada em torno de um banco de dados ou um repositório compartilhado. Dessa forma, esse tipo de arquitetura é recomendado para aplicações nas quais os dados são gerados por um componente e usados por outro, por exemplo, sistemas de informações gerenciais, sistemas de comando e controle e ambientes de desenvolvimento de software (SOMMERVILLE, 2011).



A figura 20 exemplifica uma situação na qual pode ser adotada a arquitetura de repositório, que, nesse caso, representa um IDE (*Integrated Development Environment* ou Ambiente de Desenvolvimento Integrado) que contém diversas ferramentas para apoiarem o desenvolvimento dirigido a modelos. Esse IDE contém componentes que usam um repositório de informações sobre projetos de sistema. Entretanto, esses componentes não interagem diretamente, somente por meio do repositório. Nesse exemplo, pode-se considerar que se trata de um ambiente controle de versões que mantém o controle das modificações realizadas no software e permite a retorno para versões anteriores (SOMMERVILLE, 2011).

**Figura 20 – Arquitetura de repositório para um IDE.**



**Fonte: Sommerville, 2011.**

A arquitetura de repositório pode ser adotada quando se tratar de um sistema no qual grandes volumes de informações forem gerados e precisarem ser armazenados por um longo tempo; também, poderá ser usado em sistemas dirigidos a dados, em que a inclusão dos dados no repositório dispara uma ação ou ferramenta (SOMMERVILLE, 2011).

Como vantagem dessa arquitetura, pode-se citar: (i) os componentes não precisam saber da existência de outros componentes; (ii) as alterações são feitas em um componente e podem se propagar para todos os outros; (iii) todos os dados podem ser gerenciados de forma consistente, pois tudo está em um só lugar. Em contrapartida, tem-se as desvantagens que são: (i) o fato do repositório estar em um único ponto, falhas poderão afetar todo o sistema; (ii) poderá haver ineficiências na

organização de toda a comunicação através do repositório; (iii) distribuir o repositório através de diversos computadores pode ser difícil (SOMMERVILLE, 2011).

### 2.1.3 Arquitetura Cliente-Servidor

Conforme visto na seção anterior, a arquitetura de repositório se preocupa com a estrutura estática de um software, portanto não mostra a organização do software em tempo de execução. Já a arquitetura cliente-servidor é organizada como um conjunto de serviços em que cada serviço é prestado por um servidor. Os clientes – usuários – são os que utilizam os serviços dos servidores. É, por isso, que essa arquitetura é muito utilizada para a organização de sistemas distribuídos em tempo de execução (SOMMERVILLE, 2011).

De acordo com Sommerville (2011), os principais componentes desse modelo são:

1. Conjunto de servidores. Responsáveis por oferecerem serviços a outros componentes, como servidores de impressão, servidores que gerenciam arquivos e servidores de compilação.
2. Conjunto de clientes. Tem a função de acionar os serviços oferecidos pelos servidores.
3. Rede que permite aos clientes acessar esses serviços. Grande parte dos softwares cliente-servidor é implementada como sistemas distribuídos, que são conectados por meio de protocolos de Internet.

As arquiteturas cliente-servidor são, principalmente, utilizadas para implementar serviços, como (VALENTE, 2020):

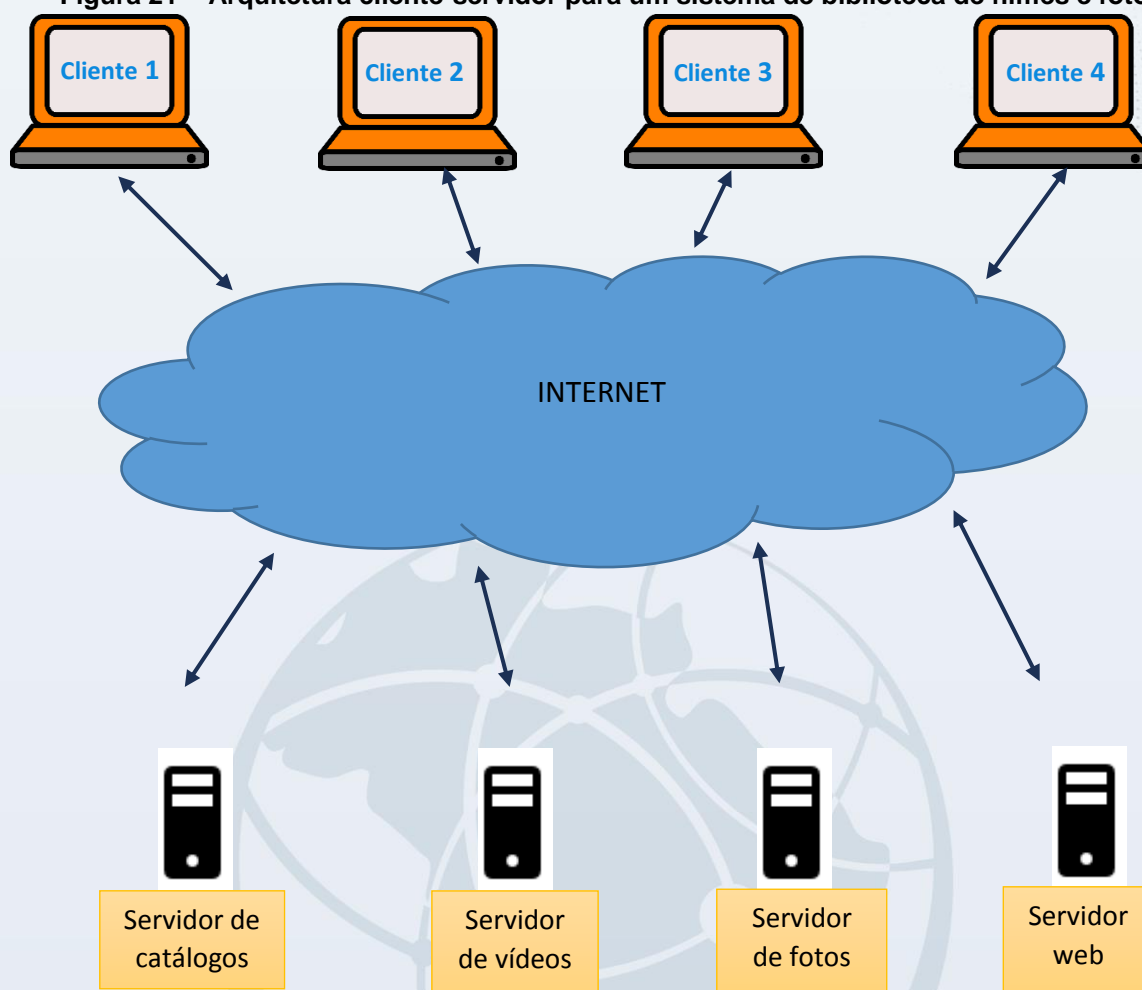
- serviço de impressão, que permitem que clientes imprimam arquivos em uma impressora remota, ou seja, uma impressora que não está conectada fisicamente ao computador dos clientes;
- serviços de arquivos, que permitem que clientes acessem o sistema de arquivos (disco) através de um computador servidor;
- serviço de banco de dados, que possibilita que os clientes acessem um banco de dados instalado em um outro computador

- e serviço Web, que consente que clientes (navegadores, por exemplo) acessem recursos (as páginas HTML) armazenadas e providas por um servidor Web.

A principal vantagem dessa arquitetura é se tratar de uma arquitetura distribuída, ou seja, os servidores podem ser disseminados por meio de uma rede de computadores. Em relação às desvantagens, pode-se citar: (i) o fato de cada serviço estar em um ponto único o torna mais suscetível à falha do servidor; (ii) o desempenho é imprevisível, pois dependerá da performance da rede de computadores; (iii) pode-se ter problemas para gerenciar os servidores se eles pertencerem a diferentes organizações (SOMMERVILLE, 2011).

A figura 21 ilustra uma arquitetura cliente-servidor para um sistema multiusuário baseado na internet, que fornece uma biblioteca de filmes e fotos. Esse sistema possui diversos servidores que gerenciam e apresentam os diferentes tipos de mídia, em que os quadros de vídeo carecem ser transmitidos de forma rápida e em sincronia, porém em resolução relativamente baixa. Já as fotos poderão permanecer em uma resolução alta, por isso é adequado conservá-las em um servidor separado (SOMMERVILLE, 2011).

Figura 21 – Arquitetura cliente-servidor para um sistema de biblioteca de filmes e fotos.



Fonte: Adaptada de Sommerville, 2011.

#### 2.1.4 Arquitetura de Duto e Filtro

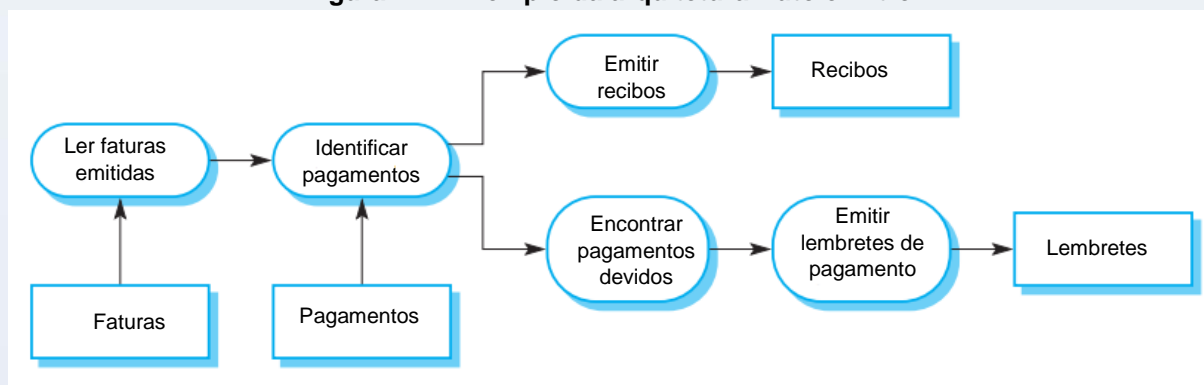
A arquitetura de duto (ou pipes) e filtro, de acordo com Sommerville (2011, pg. 114) consiste em um “modelo de organização em tempo de execução de um sistema no qual as transformações funcionais processam suas entradas e produzem saídas”.

Já Valente (2020) afirma que dutos e filtros consistem em um tipo de arquitetura orientada a dados, em que os programas são denominados de filtros, cuja função é processar os dados recebidos na entrada e reproduzir uma nova saída. Os filtros são conectados através de dutos que funcionam como buffers, ou seja, os dutos são utilizados para armazenar a saída de um filtro. Essa arquitetura é considerada bastante flexível, pois os filtros não precisam conhecer seus antecessores e sucessores, o que permite as mais variadas combinações de programas (VALENTE, 2020).



A figura 22 exemplifica a arquitetura para um sistema utilizado por um aplicativo de processamento em lote. Nesse software, os pagamentos realizados são reconciliados com as faturas e, para cada pedido pago, um recibo deverá ser emitido. Já para faturas que não foram pagas dentro do prazo de vencimento, será emitido um lembrete (SOMMERVILLE, 2011).

**Figura 22 – Exemplo da arquitetura Duto e Filtro.**



**Fonte: Sommerville, 2018.**

Geralmente, esse tipo de arquitetura é utilizado em aplicações de processamento de dados em que as entradas são processadas em fases avulsas para reproduzirem as respectivas saídas. Como vantagem dessa arquitetura, tem-se: (i) o reuso da transformação é de simples compreensão e de suporte; (ii) a estrutura de muitos processos de negócio é representado no estilo *workflows*; (iii) essa arquitetura permite tanto a implementação como um sistema sequencial quanto concorrente (SOMMERVILLE, 2011).

Já como desvantagens da arquitetura duto e filtro, tem-se: (i) o formato para transferência de dados deverá ser deliberado entre as transformações de comunicação; (ii) cada transformação deverá ser responsável por analisar sua entradas e gerar as saídas no formato estabelecido; (iii) não é adequado para sistema interativos (SOMMERVILLE, 2011).

## 2.2 Padrões de Projeto

Os padrões de projetos acarretaram grandes mudanças no projeto de software orientado a objetos, pois são soluções já testadas para problemas comuns. Em

suma, eles são uma forma de reusar o conhecimento e a experiência de outros projetistas (SOMMERVILLE, 2011).

Os padrões de projeto mais conhecidos são dos autores do livro *Design Patterns: Elements of Reusable Object-Oriented Software*. Os autores são comumente conhecidos como equipe GoF (sigla de *Gang of Four*, por serem quatro autores). Neste livro, foram descritos vinte e três padrões de projeto, conforme apresentados no quadro 3. Esses padrões estão classificados em três categorias (BEZERRA, 2015):

- Criacionais. Buscam organizar a criação dos objetos da aplicação.
- Estruturais. Organiza a aplicação de forma a facilitar a sua ampliação no futuro.
- Comportamentais. Organizam os aspectos dinâmicos da aplicação através de mecanismos de herança, agregação e composição.

Quadro 3 – Os vinte e três padrões de projeto.

Criacionais	Estruturais	Comportamentais
<ul style="list-style-type: none"> <li>✓ <b>Abstract Factory</b></li> <li>✓ <b>Builder</b></li> <li>✓ <b>Factory Method</b></li> <li>✓ <b>Prototype</b></li> <li>✓ <b>Singleton</b></li> </ul>	<ul style="list-style-type: none"> <li>✓ <i>Adapter</i></li> <li>✓ <i>Bridge</i></li> <li>✓ <i>Composite</i></li> <li>✓ <i>Decorator</i></li> <li>✓ <i>Façade</i></li> <li>✓ <i>Flyweight Proxy</i></li> </ul>	<ul style="list-style-type: none"> <li>✓ <i>Chain of Responsibility</i></li> <li>✓ <i>Command</i></li> <li>✓ <i>Interpreter</i></li> <li>✓ <i>Iterator</i></li> <li>✓ <i>Mediator</i></li> <li>✓ <i>Memento</i></li> <li>✓ <i>Observer</i></li> <li>✓ <i>Template Method</i></li> <li>✓ <i>Visitor</i></li> </ul>

Fonte: Elaborado pela autora, 2020

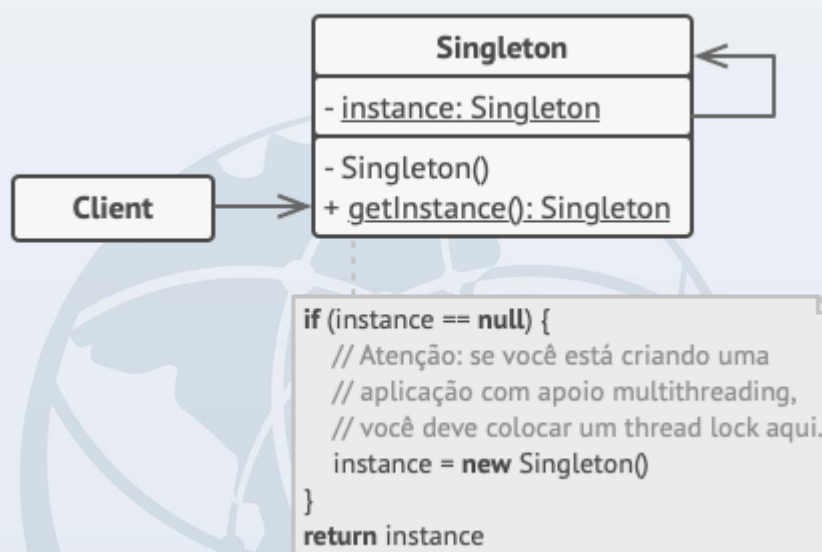
Nas próximas seções, serão apresentados três padrões de projeto dentre dos vários listados no quadro 3.

### 2.2.1 Singleton

O Singleton é um padrão criacional que assegura que uma classe tenha apenas uma instância, enquanto fornece um ponto de acesso global para essa

instância. A figura 23 exibi um exemplo de uso do Singleton. Nesse exemplo, a classe Singleton declara o método estático – *getInstance*. Esse método retorna a mesma instância de sua própria classe. O construtor da classe *Singleton* necessita ser ocultado do código da classe *Client*. Dessa forma, o único modo de obter o objeto singleton deve ser por meio do método *getInstance* (SHVETS<sup>22</sup>. Acesso em 24/06/2020).

Figura 23 - Diagrama UML com exemplo de uso do Padrão Singleton.



Fonte: SHVETS<sup>22</sup>. Acesso em 24/06/2020.

### 2.2.2 Adapter

O Adapter é um padrão de projeto estrutural que possibilita objetos com interfaces incompatíveis cooperarem entre si. A figura 24 apresenta um exemplo de uso do Adapter. Nessa implementação, é utilizado o princípio de composição do objeto – o adaptador implementa a interface de um objeto e omite o outro. *Client* é uma classe que contém a lógica de negócio do programa e a interface *Cliente Interface* define um protocolo que as outras classes devem seguir para ser capaz de contribuir com o código da classe *Client* (SHVETS<sup>23</sup>. Acesso em 24/06/2020).

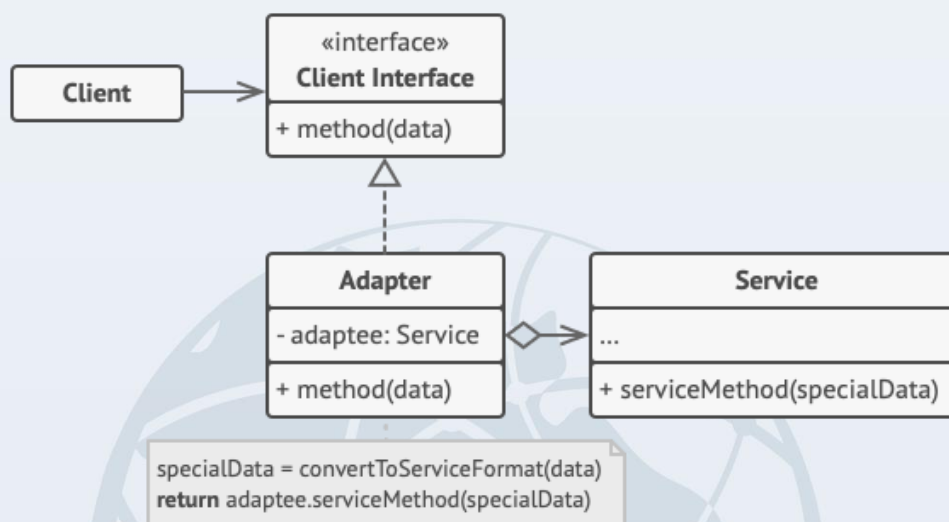
A classe *Adapter* é capaz de trabalhar tanto com o cliente quanto com o *Service*. *Service* é alguma classe útil (geralmente código de terceiros) e que o *Client*

<sup>22</sup> SHVETS, Alexander. Disponível em: <https://refactoring.guru/pt-br/design-patterns/singleton>. Acesso em 24/06/2020.

<sup>23</sup> SHVETS, Alexander. Disponível em: <https://refactoring.guru/pt-br/design-patterns/adapter>. Acesso em 24/06/2020.

não pode usá-la diretamente pelo fato de ter uma interface incompatível. O adaptador recebe chamadas da classe *Client* por meio da interface do adaptador e as traduz em chamadas para o objeto oculto do serviço em um formato que ele possa compreender (SHVETS. Acesso em 24/06/2020).

Figura 24 - Diagrama UML com exemplo de uso do Padrão Adapter.



Fonte: (SHVETS<sup>23</sup>. Acesso em 24/06/2020).

### 2.2.3 Mediator

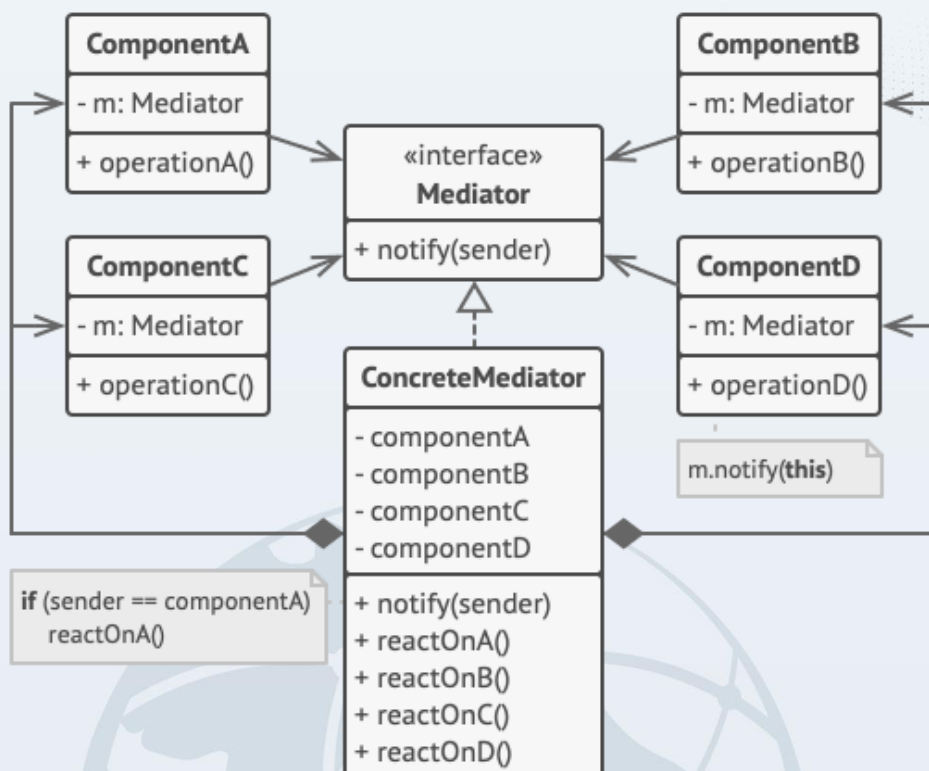
O Mediator trata-se de um padrão de projeto comportamental. Ele possibilita a redução das dependências entre objetos através da restrição das comunicações diretas entre os objetos e os força a colaborar somente por meio do objeto mediador (SHVETS<sup>24</sup>. Acesso em 24/06/2020).

A figura 25 ilustra um exemplo de uso do Mediator. Nesse exemplo, há quatro componentes: *Compenent A*, *Compenent B*, *Compenent C* e *Compenent D*. Cada um deles possui uma referência a um mediador – *interface Mediator* – portanto, nenhum deles conhece a classe *ConcreteMediator* (SHVETS<sup>24</sup>. Acesso em 24/06/2020).

<sup>24</sup> SHVETS, Alexander. Disponível em: <https://refactoring.guru/pt-br/design-patterns/mediator>. Acesso em 24/06/2020.



Figura 25 - Diagrama UML com exemplo de uso do Padrão Mediator.



Fonte: (SHVETS<sup>24</sup>. Acesso em 24/06/2020).

## INDICAÇÕES BIBLIOGRÁFICAS

GAMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

## 2.3 Implementação de Software

De acordo com Sommerville (2011, pg. 124), “o projeto e implementação de software é um estágio do processo no qual um sistema de software executável é desenvolvido”. O projeto de software é definido como uma atividade criativa que é responsável por diferenciar os componentes de software e seus relacionamentos com base nos requisitos apresentados pelo cliente. Já a implementação de software trata-se do processo de materialização do projeto como um sistema. As atividades de projeto e implementação de software são alternadas (SOMMERVILLE, 2011).

A engenharia de software abrange todas as atividades envolvidas no desenvolvimento de software, como descoberta dos requisitos, testes, manutenção e gerenciamento do sistema implantado. Entretanto, a fase mais crítica do processo

de desenvolvimento de software é a implementação. Essa fase é responsável por criar uma versão executável do software em linguagens de alto ou baixo nível, customização e adequação de sistemas genéricos para atender aos requisitos característicos de cada cliente/organização (SOMMERVILLE, 2011).

Segundo Sommerville (2011), existem três aspectos de implementação que são importantes para a engenharia de software, são eles:

1. Reuso de software. Atualmente, os novos softwares são implementados através do reuso de componentes ou sistemas. Mais detalhes serão apresentados na seção 2.3.1.
2. Gerenciamento de configuração. Este recurso é utilizado para acompanhar as diversas versões distintas geradas durante o processo de desenvolvimento do software. Mais informações poderão ser encontradas na seção 2.3.2.
3. Desenvolvimento host-target. A produção do software, normalmente, é feita em um computador denominado de *host*, e executado em outro que é nomeado de sistema *target*. Genericamente, pode-se dizer que *host* se trata de uma plataforma<sup>25</sup> de desenvolvimento e *target* de uma plataforma de execução. Ocasionalmente, as plataformas de desenvolvimento e de execução podem ser as mesmas, o que possibilita desenvolver e testar o software no mesmo computador (SOMMERVILLE, 2011). Entretanto, não é uma prática comum, sendo necessário mover o software para uma plataforma de execução para realizar os testes ou executar um simulador. Frequentemente, no desenvolvimento de softwares embutidos são utilizados simuladores, sendo possível simular dispositivos de hardwares, como sensores e os eventos no ambiente em que o sistema será implantado (SOMMERVILLE, 2011).

### 2.3.1 Reuso de Software

Atualmente, não é comum desenvolver um software a partir do zero. A abordagem baseada no desenvolvimento de reuso de softwares existentes tem ganhado cada vez mais adeptos. Pois, o reuso apresenta diversas vantagens, como

---

<sup>25</sup> Plataforma inclui o sistema operacional, softwares de apoio, como o gerenciamento de banco de dados. Caso trata-se de uma plataforma de desenvolvimento conterá um ambiente de desenvolvimento interativo.

desenvolver softwares mais rapidamente, com menos riscos de desenvolvimento e custos mais baixos devido ao fato do software reusado já ter sido testado em outras aplicações, então, provavelmente, é mais confiável que o novo software (SOMMERVILLE, 2011).

Entretanto, o reuso de software não oferece apenas vantagens, temos também algumas desvantagens, como: (i) existe uma despesa em relação ao tempo gasto para procurar e avaliar a eficiência do software a ser reusado; (ii) custos para aquisição do software a ser reusado poderá ser elevado; (iii) pode ser difícil e cara a adaptação, configuração e integração dos componentes reusáveis no software que está sendo desenvolvido (SOMMERVILLE, 2011).

É possível aplicar o reuso de software em vários níveis distintos, como (SOMMERVILLE, 2011):

1. Nível de abstração. Esse nível não se refere ao reuso do software diretamente, mas sim o conhecimento das abstrações de sucesso, como os padrões de arquitetura e de projeto (abordados nas seções: 2.1 e 2.2, respectivamente).
2. Nível de objeto. Esse nível consiste em reusar os objetos de uma biblioteca ao invés de escrever um código, como o `Arraylist`<sup>26</sup> no Java.
3. Nível de componentes<sup>27</sup>. Esse nível trata-se do reuso do componente que é utilizado para construir a interface de usuário através de um *framework*. Um *framework* consiste em um conjunto de classes de objetos que implementam a manipulação de eventos, gerenciamento de displays e etc. Posteriormente, o desenvolvedor adiciona as conexões com os dados a serem expostos e implementa o código para definir os detalhes específicos do display, como o layout da tela e cores.
4. Nível de sistema. Nesse nível, o desenvolvedor reusa todo o sistema de aplicação. Normalmente, requer algum tipo de configuração. Elas podem ser feitas por meio da adição e modificação do código, como quando está reusando uma linha de produtos de software ou pelo uso de interface de

<sup>26</sup> `Arraylist` é uma classe para coleção.

<sup>27</sup> Componentes consistem em coleções de objetos e classes de objetos que atuam em conjunto para fornecer funções e serviços relacionados.

configuração do próprio sistema. A grande maioria dos softwares são criados dessa forma.

### 2.3.2 Gerenciamento de Configuração

O gerenciamento de configuração do software trata-se de um processo para gerenciar as mudanças que ocorrem no software. O objetivo desse processo é apoiar a integração do software facilitando para que todos os desenvolvedores possam acessar o código do projeto e os documentos relacionados de forma controlada. Pois, é necessário garantir que os membros da equipe de desenvolvimento não façam alterações sobre os trabalhos dos outros membros. Além disso, é preciso assegurar que todos os desenvolvedores acessem as versões mais atuais do software, caso contrário poderá refazer um trabalho que já foi feito (SOMMERVILLE, 2011). Outra vantagem é que se algo der errado em alguma versão é possível voltar a versão anterior ou a mais estável.

De acordo com Sommerville (2011) existem três atividades fundamentais no gerenciamento de configuração de software, que são:

1. Gerenciamento de versões. Consiste no suporte fornecido para manter o controle das diferentes versões dos componentes do software. As ferramentas que permitem o gerenciamento de versões, como Subversion<sup>28</sup>, Git<sup>29</sup> e mercurial<sup>30</sup>, contêm recursos para registrar toda a evolução do software e manter armazenado todas as alterações realizados no código.
2. Integração de sistemas. Trata-se do suporte fornecido aos desenvolvedores que os ajuda a definir quais são as versões dos componentes que colaboraram para a criação de cada versão do software. Essas informações são utilizadas para construir um sistema automaticamente, compilando e conectando aos componentes essenciais. Uma ferramenta bastante conhecida para integração de sistemas é o Jenkins<sup>31</sup>.

<sup>28</sup> <https://subversion.apache.org/>

<sup>29</sup> <https://git-scm.com/>

<sup>30</sup> <https://www.mercurial-scm.org/>

<sup>31</sup> <https://www.jenkins.io/>



3. Rastreamento de problemas. Oferece suporte para auxiliar os desenvolvedores a informar o bugs e outros problemas e ver como eles foram solucionados. Bugzilla<sup>32</sup> é uma ferramenta muito utilizada pelos desenvolvedores para essa finalidade.

## 2.4 Teste de Software

Segundo Pressman (2011, pg. 401), “teste é um conjunto de atividades que podem ser planejadas com antecedência e executadas sistematicamente. Por essa razão, deverá ser definido para o processo de software um modelo (*template*) para o teste”. Já para Sommerville (2011, pg. 144), “o teste é destinado a mostrar que um programa faz o que é proposto a fazer e para descobrir os defeitos do programa antes do uso”.

O teste de software consiste em um amplo processo de verificação e validação (V&V). A *verificação* trata-se de um conjunto de tarefas que garantem que o software implementa corretamente uma determinada função, ou seja, verifica se o software atende os requisitos funcionais e não funcionais. Já a *validação* fundamenta-se em um conjunto de tarefas que certifica que o software foi desenvolvido e pode ser validado conforme os requisitos do cliente, ou seja, pretende comprovar que o software faz o que o cliente espera que ele faça (SOMMERVILLE, 2011).

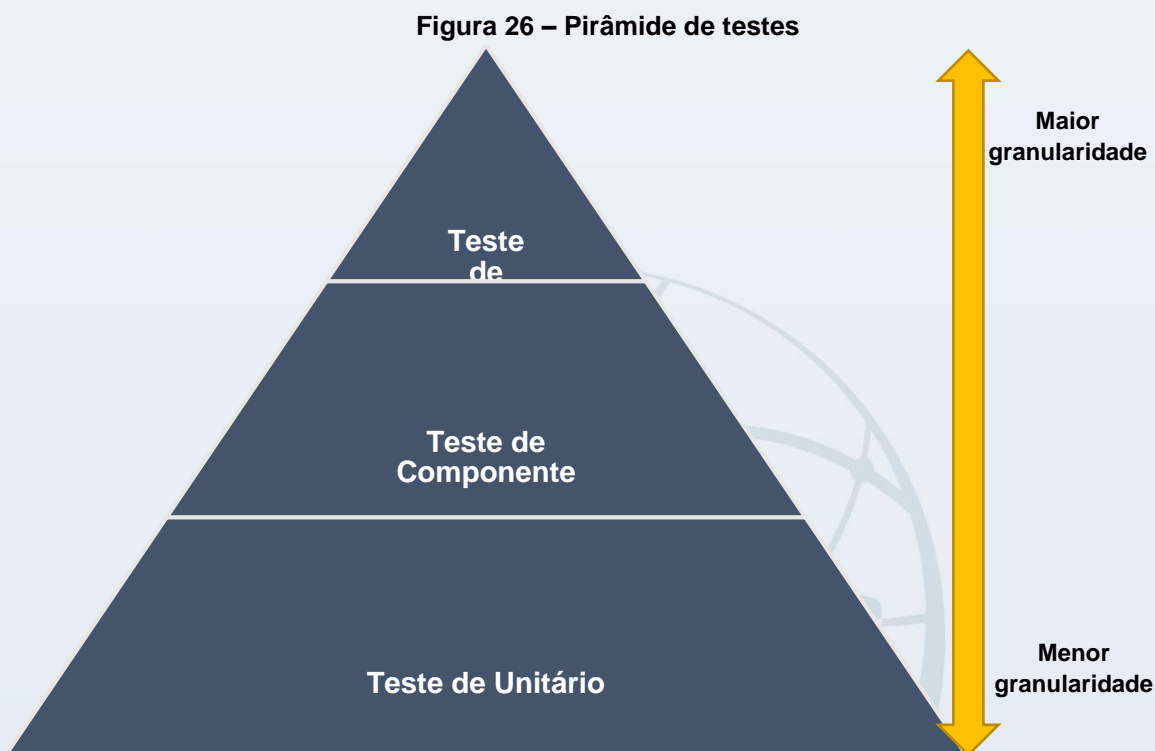
Geralmente, o processo de teste envolve uma mistura de testes manuais e automatizados. O teste manual trata-se daquele que é executado por um testador, sendo ele o responsável por executar o software com alguns dados de teste. Posteriormente, ele compara os resultados obtidos com os resultados esperados, em seguida o testador produz um relatório de testes relatando as discrepâncias encontradas aos desenvolvedores. Já nos testes automatizados, os testes são codificados por meio de um framework, como o JUnit. Dessa forma, o teste automático é executado cada vez que o sistema em desenvolvimento necessita ser testado. O teste automatizado é bem mais rápido que o teste manual, principalmente, se houver necessidade de realizar testes de regressão – reexecução

---

<sup>32</sup> <https://www.bugzilla.org/>

dos testes feitos anteriormente para averiguar se as modificações no software não introduziram novos erros (SOMMERVILLE, 2011).

Segundo Sommerville (2011), durante o desenvolvimento do software, o teste pode possuir até três níveis de granularidade que estão representados na figura 26.



Fonte: Elaborado pela autora, 2020.

Os testes também podem ser classificados conforme as técnicas de teste que são: caixa preta (*black box*) ou caixa-branca (*white box*). A técnica caixa-preta é também denominada de testes funcionais. Essa técnica refere-se aos testes que são escritos baseados na interface do sistema, por exemplo, se o objetivo for testar um método as únicas informações disponíveis serão: nome, parâmetros, tipos e exceções de retorno. Já a técnica caixa-branca, que é chamada de testes estruturais, trata-se da escrita dos testes que considera informações sobre o código e a estrutura do sistema sobre o teste (VALENTE, 2020).

**SAIBA MAIS**

Assunto: Introdução a Testes

Disponível

em:

<[https://edisciplinas.usp.br/pluginfile.php/3503764/mod\\_resource/content/3/Introducao\\_a\\_Testes\\_de\\_Software.pdf](https://edisciplinas.usp.br/pluginfile.php/3503764/mod_resource/content/3/Introducao_a_Testes_de_Software.pdf)>.

**2.4.1 Teste Unitário**

O teste unitário é também conhecido como teste de unidade, cuja a função é testar pequenas unidades de código, geralmente, são métodos ou classes de objetos, em que são testados de forma isolada do restante do software. A principal vantagem do teste unitário é localizar *bugs* durante a fase de desenvolvimento do software, ou seja, antes que o código entre em produção, fase na qual os custos de correção e prejuízos podem ser altos (VALENTE, 2020).

Na maior parte dos casos, deve-se automatizar os testes unitários através de algum *framework*, como o JUnit, para criar e executar testes no software. De acordo com Sommerville (2011), um teste automatizado possui três partes, sendo elas:

- Parte de configuração que serve para iniciar o software com o caso de teste, isto é, as entradas e saídas almejadas.
- Parte de chamada acontece quando se chama o objeto ou método a ser testado.
- Parte de afirmação trata-se da parte que os resultados da chamada são comparados ao resultado esperado. Se o resultado for avaliado como verdadeiro, significa que o teste foi bem-sucedido, caso seja falso indica que o teste falhou.

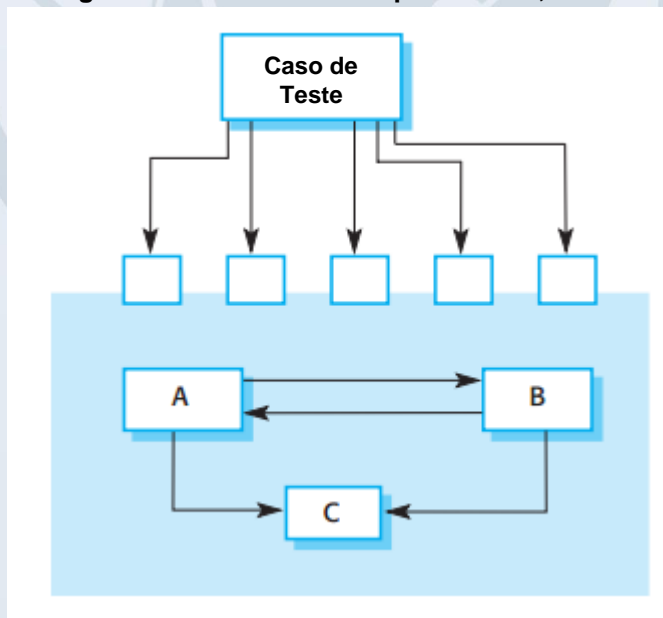
Ocasionalmente, poderá haver a necessidade de testar objetos que possuem dependências a outros objetos que ainda não foram implementados ou que atrasam o processo de teste quando são utilizados; nesse caso, pode-se utilizar um *mock object*. *Mock objects* consistem em objetos que tem a mesma interface que os objetos externos utilizados para simular sua funcionalidade. Por exemplo, um *mock object* poderá ser utilizado para simular um banco de dados com uns pouco elementos organizados em um vetor. Ou então, podem ser utilizados para simular

operações anormais ou eventos raros (SOMMERVILLE, 2011). Existem frameworks que facilitam a criação e programação de *mocks objects*, dentre elas pode-se citar o Mockito<sup>33</sup>.

#### 2.4.2 Teste de Componente

No teste de componente, várias unidades individuais são integradas para criar componentes compostos. Esse tipo de teste concentra-se em mostrar que as interfaces dos componentes se comportam conforme a sua especificação. A figura 27 exemplifica a ideia de teste de interface de componentes. Nessa figura, existem três componentes: A, B e C que foram integrados para produzir um componente maior. Nesse exemplo, os testes não serão mais aplicados aos componentes individuais, mas sim a interface produzida pela combinação dos componentes A, B e C. O teste unitário não consegue identificar os erros ocorridos após a combinação dos componentes, pois eles são originados das interações entre os objetos do componente (SOMMERVILLE, 2011).

Figura 27 – Teste dos Componentes A, B e C.



Fonte: Sommerville, 2018.

<sup>33</sup> <https://site.mockito.org/>



### 2.4.3 Teste de sistema

De acordo com Sommerville (2011, pg. 153), o teste de sistema tem a função de verificar “se os componentes são compatíveis, se interagem corretamente e transferem os dados certos no momento certo, por suas interfaces”. Ou seja, simulam o uso do software por um usuário real. Existem vários frameworks para automatizar esse tipo de teste, dentre eles tem-se o Selenium<sup>34</sup>. O Selenium possibilita criar softwares que funcionam como robôs; abrindo páginas Web, preenchendo formulários, testam respostas e até clicam em botões (VALENTE, 2020). Normalmente, é o tipo de teste mais difícil de se criar quando comparado aos testes automatizados de unidades ou componentes.

Em suma, o teste de sistema concentra-se em testar as interações entre os componentes e objetos que constituem o software. Por sua vez, o teste de interação serve para: (i) descobrir *bugs* de componentes que só ocorrem quando um componente é utilizado por outros componentes do software e (ii) localizar equívocos dos desenvolvedores sobre outros componentes do software (SOMMERVILLE, 2011).

### 2.4.4 Teste de usuário

O teste de usuário é também denominado de teste de cliente. O teste de usuário é uma fase no processo de teste em que clientes/stakeholders fornecem entradas e opiniões sobre o teste de sistema (SOMMERVILLE, 2011).

Segundo Sommerville (2011), existem três tipos de testes de usuário, são eles:

- Teste alfa. Trata-se de um tipo de teste em que os usuários e desenvolvedores trabalham juntos para testar o software no ambiente do desenvolvedor.
- Teste beta. Consiste em fornecer uma versão (release) do software que está sendo desenvolvido aos usuários para que possam experimentá-lo e reportar aos desenvolvedores os problemas encontrados.
- Teste de aceitação. Consiste em um teste realizado pelo cliente, cuja o objetivo é ele decidir se aceita ou não o software, ou seja, se ele está bom o suficiente para ser implantado e utilizado no ambiente do cliente. O

<sup>34</sup> <https://www.selenium.dev/documentation/en/>

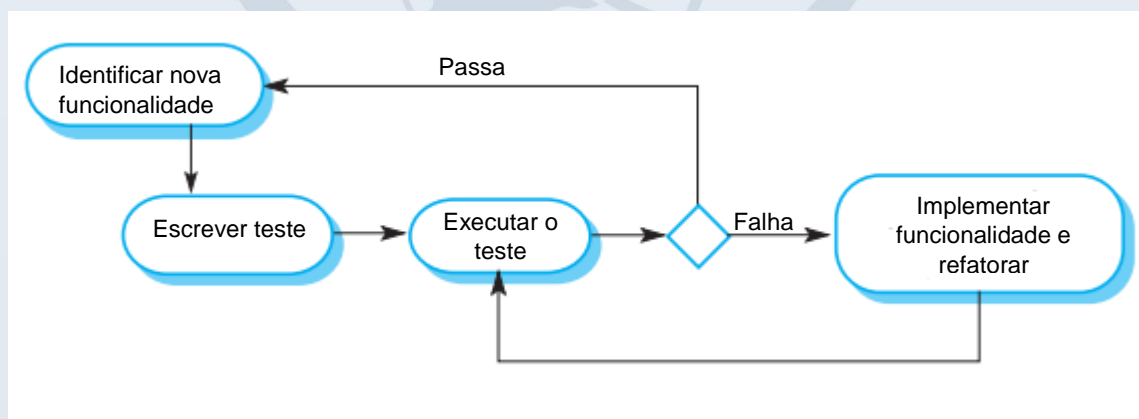
teste de aceitação tem por verificar se tanto os requisitos funcionais como os não funcionais foram cumpridos. Porém, muitas das vezes, é difícil estabelecer critérios objetivos de aceitação.

#### 2.4.5 Desenvolvimento dirigido a testes

O desenvolvimento dirigido a testes (*Test-Driven Development – TDD*) consiste em uma metodologia para desenvolver softwares em que é intercalado testes e o desenvolvimento de código. Esse tipo de teste consiste em uma das práticas de programação propostas pelo método ágil XP (*eXtreme Programming*) (SOMMERVILLE, 2011). No entanto, o TDD pode ser também utilizado no processos de desenvolvimento de software que adotam a metodologia tradicional. De acordo com Valente (2020), o TDD foi proposto com os seguintes objetivos: (i) evitar que desenvolvedores esqueçam-se de escrever testes; (ii) favorecer a escrita de código com alta testabilidade.

A ideia básica do TDD é escrever o teste, localizar uma falha durante o mesmo teste e, posteriormente, refatorá-lo. Em suma, consiste na repetição de um ciclo de desenvolvimento concentrado em testes unitários. Sommerville (2011) define cinco etapas do processo básico de TDD, as quais são ilustradas na figura 28.

Figura 28 – Exemplificação das etapas do TDD.



Fonte: Sommerville, 2018.

A primeira etapa consiste na identificação da nova funcionalidade, ou seja, identificar o incremento de funcionalidade necessário, que deverá ser implementado

em poucas linhas de código. A segunda etapa trata-se da escrita de teste para essa funcionalidade e implementar um teste automatizado (SOMMERVILLE, 2011).

Posteriormente, tem-se a terceira etapa que é a execução do teste. Durante a execução desse teste, que é feito juntamente com todos os outros testes implementados. Como, inicialmente, não haverá todas as funcionalidades implementadas, logo o teste falhará. Mas, na quarta etapa (Falha), esse problema será sanado. Pois, nela é implementado a funcionalidade faltante e, novamente, é executado os testes. Após os resultados do teste, poderá haver a necessidade de refatorar<sup>35</sup> o código para melhorá-lo e/ou adicionar um novo código. E, por último, a quinta etapa que é a implementação de funcionalidade e refatoração. Nessa etapa, após ter alcançado sucesso em todos os testes, poderá implementar a próxima parte da funcionalidade (SOMMERVILLE, 2011).

#### SAIBA MAIS

Assunto: TDD - Desenvolvimento Orientado a Testes

Disponível

no

link:

<<http://revista.universo.edu.br/index.php?journal=1reta2&page=article&op=download&path%5B%5D=1402&path%5B%5D=1037>>.

## 2.5 Suporte e Manutenção do Software

A manutenção do software trata-se de um processo geral para a realização de modificações em um software após ele ser liberado para uso. Essas modificações podem ser simples, como correção de erros de codificação ou mudanças maiores para correção de erros de projeto ou melhorias significativas para corrigir erros de especificação ou então inclusão de novos requisitos. As alterações no código são implementadas por meio da alteração de componentes e também da inclusão de novos componentes no software.

De acordo com Sommerville (2018), existem três tipos distintos de manutenção de software, são eles:

<sup>35</sup> Consiste em modificar um software ou trecho dele para melhorar sua estrutura, reduzir sua complexidade ou facilitar o entendimento.

- Manutenção Corretiva ou Correção de Defeitos. Consiste em um tipo de manutenção para corrigir os erros e vulnerabilidades do software. É importante salientar que os erros de codificação são simples de serem corrigidos. Já os erros de requisitos que são os mais caros para se corrigir, pois um extenso reprojetado do software poderá ser necessário.
- Manutenção Adaptativa ou Adaptação Ambiental. Esse tipo de manutenção é necessário quando o software carece de alguma adaptação, como mudanças nas regras de negócio, leis que tenham consequências a funções do sistema ou mesmo uma adaptação para atender a uma nova versão de um sistema operacional que venha a não ser totalmente compatível ao produto de software.
- Manutenção Perfectiva ou Evolutiva. Trata-se de um tipo de manutenção que tem o papel de aperfeiçoar o software, ou seja, melhorar a qualidade do software. Esse aperfeiçoamento poderá ocorrer através da adição ou de modificação das funcionalidades ao software. Esse tipo de manutenção, geralmente, é necessário quando os requisitos do software precisam ser modificados para atender às mudanças organizacionais ou de negócios.

Segundo Sommerville (2018), na prática, não há uma distinção clara sobre os três tipos de manutenção de software. Pois, ao realizar a adaptação do software a um novo ambiente poderá também adicionar funcionalidades para tirar proveito dos novos recursos ambientais. Frequentemente, as falhas de software são mais expostas pelo fato dos usuários usarem-no de maneiras imprevistas. A alteração do software para adaptar-se a maneira de trabalhar dos usuários que usam o sistema é a melhor forma de corrigir essas falhas.

### 2.5.1 Suporte a Software

Muitas empresas ao comprar ou adquirir uma licença de uso de um software consideram apenas o valor mais baixo dessa aquisição. Frequentemente, não avaliam a necessidade de contratar também um serviço de suporte do software. Em



um primeiro momento pode parecer irrelevante esse tipo de serviço e, portanto, o pensamento de que vale a pena economizar prevalece.

Entretanto, se a empresa não adquire esse tipo de serviço ela precisará investir um setor próprio de TI (tecnologia da informação) ou consumir tempo e recursos na busca de prováveis soluções, o que poderá acarretar em grandes prejuízos. Diante dos problemas expostos, recomenda-se que a empresa considere adotar um software que forneça um suporte completo. Dentre as vantagens da contratação do suporte, pode-se citar:

- Oferecimento de treinamento necessário para que os gestores e/ou colaboradores da empresa conheçam profundamente a potencialidade e as especialidades do software. Dessa forma, o software poderá ser utilizado de um modo mais assertório.
- Garantia que haverá atualizações, como as de melhorias dos processos da empresa para os tornar mais eficiente. Pois, a empresa deve avaliar e pensar que grande parte das empresas concorrentes estão sempre em busca de melhorias para tornar seus processos mais eficientes e aumentar a produtividade e, consequentemente, tornarem-se mais competitivas no mercado.
- O suporte ser feito pelos próprios desenvolvedores do software, que detêm de total conhecimento sobre ele, torna mais fácil, rápido identificar e resolver possíveis problemas antes que eles causem prejuízos.

O suporte técnico poderá ocorrer de duas formas: (i) remoto, que ocorre através de e-mail, telefone, chat e uso de ferramentas de acesso remoto (com a ferramenta VNC); ou (ii) presencial que é também denominado de *supor in loco*. Em suma, o suporte a software tem uma importância fundamental tanto para qualquer empresa que utiliza um software para auxiliar na sua gestão.

### 2.5.2 Treinamento de Software

O treinamento de software é fundamental para que o cliente/*stakeholder* consiga explorar ao máximo todo potencial do software que foi adquirido. Esse treinamento não precisa ocorrer apenas presencialmente. Entretanto, o fato do

treinamento ser presencial aproxima a relação entre empresa e cliente/*stakeholder* e também facilita a percepção quanto às dúvidas deles. Porém, o treinamento presencial costuma ter um custo bastante elevado, pois existem deslocamento e o investimento de horas dedicadas do profissional (MELO, 2017).

Uma solução para esse custo elevado do treinamento presencial é mesclá-lo com o ensino a distância. Dessa forma, pode-se dar a parte teórica do software à distância (on-line) e a prática – utilização do software – ser feita presencialmente. Um outro meio de realizar o treinamento é através de uma conferência com compartilhamento de tela. Essa alternativa possibilita a proximidade com o cliente/*stakeholder* e também a resolução imediata das dúvidas e com a vantagem de ser mais barato, pelo fato de ser feito a distância. Existem diversas ferramentas que possibilitam o compartilhamento de telas durante uma conferência (chamada), como o Skype e o Hangout (MELO, 2017).

### 2.5.3 Documentação do Software

A documentação de um software, geralmente, não é uma prioridade para muitas empresas que desenvolvem sistemas, seja pelo fato dos colaboradores preferirem codificar a documentar, ou então, simplesmente, por acreditarem que documentação é perda de tempo.

Existem vários tipos de documentação de software, dentre elas: (i) para usuários técnicos que são os desenvolvedores, testadores e projetistas e (ii) usuários finais que são as pessoas que irão usar o software. A documentação em ambos os casos é importante para auxiliar na redução do tempo de compreensão e aprendizagem dos usuários técnicos e finais. Principalmente, para quando houver a necessidade de evoluções no software (GUEDES, 2019).

A documentação do software é vista com uma fonte de informações do software, pelo fato de possuir todas as operações e informações necessárias do software, como trechos de código, diagramas UML, definições de requisitos, especificação do software, planos de testes, dentre outros que se façam necessários (GUEDES, 2019).

## CAPÍTULO 3 – MELHORIA CONTÍNUA DO SOFTWARE

### 3.1 Modelo CMMI

O instituto de Engenharia de Software (*Software Engineering Institute*, SEI) da Carnegie Mellon University (EUA) foi criado mirando a melhoria das capacidades da indústria norte-americana de software. Em 1980, o SEI criou o Modelo Maturidade e de Capacidade (*Capability Maturity Model*, CMM), cuja o objetivo seria avaliar as capacidades dos prestadores de serviços de software (SOMMERVILLE, 2011).

Na década de 1990, o SEI ampliou o modelo CMM para o modelo de capacidade integrado, que é o CMMI (*Capability Maturity Model Integration*). Pressman (2011, pg. 690) define o CMMI como “um metamodelo de processo abrangente qualificado em uma série de capacidades de sistema e engenharia de software que devem estar presentes à medida que as organizações alcançam diferentes níveis de capacidade e maturidade de processo”.

Existem três modelos diferentes baseados no modelo CMMI (LINSTEDT and OSLCHIMKE, 2016. s.p, Tradução nossa)<sup>36</sup>:

- CMMI for Development (CMMI-DEV) consiste em um modelo de processo para gerenciamento e melhoria de processos em organizações de desenvolvimento de software.
- CMMI for Acquisition (CMMI-ACQ) trata-se de um modelo para organizações que precisam iniciar e gerenciar a aquisição de produtos e serviços.
- CMMI for Services (CMMI-SVC) refere-se a um modelo de processo para organizações para ajudá-las a implantar e gerenciar serviços.

O CMMI representa o metamodelo de duas formas: modelo por estágios e modelo contínuo. O CMMI permite que o desenvolvimento de softwares e processos de gerenciamento de uma empresa seja avaliado e, posteriormente, receba um nível

<sup>36</sup> *There are three different models based on the CMMI framework [7]: CMMI for Development, a process model for process management and improvement in software development organizations. CMMI for Acquisition, a model for organizations that have to initiate and manage the acquisition of products and services. CMMI for Services, a process model for organizations to help them to deploy and manage services.*



de maturidade que varia de 1 a 5. O CMMI na forma contínua possibilita uma classificação de granularidade mais baixa de maturidade de processo; também fornece uma forma de classificar vinte e duas áreas de processo em uma escala de 0 a 5 (SOMMERVILLE, 2011). O quadro 4 apresenta essas 22 áreas de processo no CMMI.

**Quadro 4 – Áreas de processo**

<b>Categoria</b>	<b>Área de processo</b>
Gerenciamento de Processo	<ol style="list-style-type: none"> <li><b>1. Definição de processo organizacional</b> (OPD, <i>Organization Process Definition</i>)</li> <li><b>2. Foco de processo organizacional</b> (OPF, <i>Organization Process Focus</i>)</li> <li><b>3. Treinamento organizacional</b> (OT, <i>Organization Focus</i>)</li> <li><b>4. Desempenho de processo organizacional</b> (OPP, <i>Organization Process Performance</i>)</li> <li><b>5. Inovação e Implantação organizacional</b> (OID, <i>Organization Innovation and Deployment</i>)</li> </ol>
Gerenciamento de Projeto	<ol style="list-style-type: none"> <li><b>1. Planejamento de projeto</b> (PP, <i>Project Planning</i>)</li> <li><b>2. Monitoração e controle de projeto</b> (PMC, <i>Project Monitoring na Control</i>)</li> <li><b>3. Gerenciamento de acordo com fornecedores</b> (SAM, <i>Supplier Agreement Management</i>)</li> <li><b>4. Gerenciamento de projeto integrado</b> (IPM, <i>Integrated Project Management</i>)</li> <li><b>5. Gerenciamento de riscos</b> (RSKM, <i>Risk Management</i>)</li> <li><b>6. Gerenciamento quantitativo de projeto</b> (QMP, <i>Quantitative Project Management</i>)</li> </ol>
Engenharia	<ol style="list-style-type: none"> <li><b>1. Gerenciamento de requisitos</b> (REQM, <i>Requeriments Management</i>)</li> <li><b>2. Desenvolvimento de requisitos</b> (RD, <i>Requeriments Development</i>)</li> <li><b>3. Solução técnica</b> (TS, <i>Technical Solution</i>)</li> <li><b>4. Integração de produto</b> (PI, <i>Product Integration</i>)</li> <li><b>5. Verificação</b> (VER, <i>Verification</i>)</li> <li><b>6. Validação</b> (VAL, <i>Validation</i>)</li> </ol>
Suporte	<ol style="list-style-type: none"> <li><b>1. Gerenciamento de configuração</b> (CM, <i>Configuration Management</i>)</li> <li><b>2. Garantia de qualidade de processo de produto</b> (PPQA, <i>Process and Product Quality Management</i>)</li> </ol>



	<p><b>3. Medição e análise</b> (MA, <i>Measurement and Analysis</i>)</p> <p><b>4. Análise de decisão e resolução</b> (DAR, <i>Decision Analysis and Resolution</i>)</p> <p><b>5. Análise causal e resolução</b> (CAR, <i>Causal Analysis and Resolution</i>)</p>
--	--

Fonte: Adaptação de Sommerville, 2011.

De acordo com Sommerville (2011), um conjunto de boas práticas consistem em definições de meios de como alcançar uma meta. Diversas práticas específicas e genéricas, que são apresentadas no quadro 5, podem estar associadas com cada meta dentro de uma área de processo.

**Quadro 5 – Metas e práticas associadas no CMMI.**

<b>Metas</b>	<b>Práticas associadas</b>
<i>Os requisitos são analisados e validados e uma definição da funcionalidade requerida é desenvolvida.</i>	<ul style="list-style-type: none"> <li>• Analisar sistematicamente os requisitos derivados para garantir que eles são necessários e suficientes.</li> <li>• Validar os requisitos para garantir que o produto resultante executará conforme pretendido no ambiente do usuário, usando várias técnicas conforme apropriado.</li> </ul>
<i>Causas-raiz de defeitos e outros problemas são sistematicamente determinados.</i>	<ul style="list-style-type: none"> <li>• Selecionar os defeitos críticos e outros problemas para análise.</li> <li>• Realizar uma análise casual de defeitos selecionados e outros problemas e propor ações para solucioná-los.</li> </ul>
<i>O processo é institucionalizado como um processo definido.</i>	<ul style="list-style-type: none"> <li>• Estabelecer e manter uma política organizacional para planejar e executar o processo de desenvolvimento de requisitos.</li> <li>• Atribuir responsabilidade e autoridade para executar o processo, desenvolver os produtos de trabalho e prestar os serviços do processo de desenvolvimento de requisitos.</li> </ul>

Fonte: Sommerville, 2011.

De acordo com Pressman (2011), cada área do processo, como gerenciamento de requisitos, é formalmente avaliada em relação a alvos e práticas específicas e classificada conforme os seguintes níveis de capacidade que vão do 0 ao 5:

- Nível 0: Incompleto – Alguma área de processo, como a gerenciamento de requisitos, não funciona ou não abrange todas as metas e objetivos definidos pela CMMI para alcançar o nível 1 (PRESSMAN, 2011).
- Nível 1: Executado – Todas as metas específicas da área de processo do modelo CMMI foram realizadas e estão sendo realizadas as tarefas necessárias para produzir os artefatos definidos (PRESSMAN, 2011).
- Nível 2: Controlada (Gerenciado) – Nesse nível todos os critérios do nível 1 foram contentados e todo o trabalho associado com a área de processo está de acordo com uma política definida em termos de organização. Além disso, todos os trabalhadores têm acessos aos adequados recursos para executar o trabalho e os interessados são envolvidos ativamente na área de processo conforme necessário (PRESSMAN, 2011). “Todas as tarefas e produtos são monitorados, controlados e revisados; e são avaliados quanto a conformidade com a descrição de processo”.
- Nível 3: Definido – Nesse nível, todos os critérios do nível anterior (nível 2) foram contentados. O nível 3 concentra-se na padronização e implantação de processos organizacionais e cada projeto tem um processo gerenciado que é ajustado aos requisitos de projeto de um conjunto definido de processos organizacionais (SOMMERVILLE, 2011).
- Nível 4: Controlado quantitativamente – Nesse nível, todos os critérios do nível 3 devem ter sido atendidos. No nível 4, a área de processo é controlada e melhorada usando medição e avaliação quantitativa (PRESSMAN, 2011).
- Nível 5: Otimizando – Nesse nível, todos os critérios do nível 4 foram atendidos e a área de processo é adaptada e otimizada usando meios quantitativos (estatísticos) para atender à mudança de necessidade do cliente e melhorar continuamente a eficiência da área de processo em consideração (PRESSMAN, 2011).

A definição apresentada anteriormente para o níveis de capacidade do CMMI é muito simplificada. Pois, o modelo é muito complexo e extenso com cerca de mais de mil páginas de descrição. Portanto, na prática há a necessidade de trabalhar com a descrição mais detalhada que apresentada nessa apostila. Conforme se pôde observar, o níveis de capacidade são progressivos, portanto, para uma empresa melhorar seus processos, ela deverá buscar aumentar o nível de maturidade dos grupos de processos que são relevantes para o seu negócio (SOMMERVILLE, 2011).

De acordo com Pressman (2011, pg. 693), “CMMI é uma conquista significativa na engenharia de software”. Pelo fato de proporcionar uma discussão das atividades e ações que devem existir quando uma empresa desenvolve softwares (PRESSMAN, 2011).

### 3.1.1 Modelo CMMI por estágios

O modelo CMMI por estágios possibilita uma forma de avaliar a capacidade de processo de uma empresa em um dos cinco níveis e determina as metas que devem ser alcançadas em cada um desses níveis. A melhoria de processos é alcançada por meio da implementação de práticas em cada nível, desde os níveis baixos até os níveis mais altos do modelo por estágios (SOMMERVILLE, 2011). Esse modelo possui cinco estágios que estão ilustrados nas figura 29.

Figura 29 – Modelo de maturidade CMMI por estágios



Fonte: Elaborada pela autora, 2020.

A vantagem do modelo CMMI por estágios é ele ser compatível com o modelo CMM, pois muitas empresas compreendem e comprometem a usar esse modelo para melhorar os processos de software. Sendo assim mais fácil migrar para o modelo CMMI por estágios que define um caminho de melhoria clara para as organizações. Elas podem planejar avançar para o segundo, terceiro nível e assim sucessivamente (SOMMERVILLE, 2011).

No entanto, também há desvantagem: esse modelo pressupõe que todas as metas e práticas em um nível sejam praticadas antes da transição para o próximo nível. Porém, na prática, pode ser mais adequado implementar as metas e práticas em níveis superiores antes das práticas de nível inferiores e, quando uma empresa faz isso, a avaliação da maturidade dará uma falsa imagem de sua capacidade (SOMMERVILLE, 2011).

### 3.1.2 Modelo CMMI Contínuo

O modelo CMMI contínuo consistem em modelos mais refinados que consideram avaliam o uso de boas práticas dentro de cada grupo de processo individualmente. Ao invés de classificar uma organização em níveis discretos. Dessa forma, as empresas operam em diferentes níveis de maturidade para cada processo ou grupo de processos (SOMMERVILLE, 2011).

O modelo CMMI contínuo pondera algumas das áreas de processo apresentadas no quadro 5 (SOMMERVILLE, 2011), por exemplo:

- Planejamento de Projeto (PP).
- Gerenciamento de Requisitos (REQM).
- Mediação e Análise (MA).
- Gerenciamento de Configuração (GM).
- Garantia de Qualidade de Processo e Produto (PPQA).

Posteriormente, atribui um nível de avaliação de capacidade em uma escala de seis níveis para cada área de processo, que são (SOMMERVILLE, 2011): (i) Nível 0 - Incompleto; (ii) *Nível 1 - Executado*; (iii) *Nível 2 - Gerenciado*; (iv) *Nível 3 - Definido*; (v) *Nível 4 - Quantitativamente gerenciado*; e (vi) *Nível 5 - Otimizando*. Esses níveis já foram conceituados na seção 3.1.



A figura 30 ilustra um perfil de capacidade que exhibe os processos em níveis díspares de capacidade associada. Pela figura é possível observar que o nível de maturidade em gerenciamento de configuração (REQM) é mais alto que a maturidade do Mediação e Análise (MA). Uma organização pode desenvolver perfis de capacidade reais e também um perfil almejado, tornando possível ela visualizar qual nível de capacidade a empresa deseja atingir para essa área de processo (SOMMERVILLE, 2011).

A principal vantagem do modelo CMMI contínuo é ser flexível, dessa forma a organização pode decidir quais áreas de processos devem ser aperfeiçoadas (SOMMERVILLE, 2011). Por exemplo, uma organização que desenvolve softwares pode optar por se concentrar em melhorias no gerenciamento de requisitos. Já o modelo por estágios é utilizado para avaliar a capacidade da empresa como um todo, ou seja, obriga a organização a se concentrar em diferentes estágios e não só no qual ou quais são mais importantes para ela como ocorre no modelo CMMI contínuo.

**Figura 30 – Modelo de maturidade CMMI contínuo**



**Fonte: Adaptada de Pressman, 2011.**

**SAIBA MAIS**

Assunto: Capability Maturity Model Integration

Disponível no link: <<https://www.sciencedirect.com/topics/computer-science/capability-maturity-model-integration>>.

**3.2 Modelo de rocesso do Software Brasileiro (MPS.BR)**

O Modelo de Processo do Software Brasileiro (MPS.BR) foi criado em 2003 pela empresa Softex (Associação para Promoção da Excelência do Software Brasileiro), algumas outras empresas e instituições de ensino. O movimento de criação do MPS.BR também teve participação do Ministério da Ciência, Tecnologia e Inovação.

O MPS.BR possui compatibilidade com também as normas ISO (ISO/IEC 12207:2008, ISO/IEC 15504, ISO/IEC 20000) e duas variações do modelo CMMI (CMMI-DEV e CMMI-SVC) que contribuíram para a base técnica da construção e aperfeiçoamento do MPS.

Ele tem como objetivo estabelecer e aperfeiçoar um modelo de melhoria e avaliação de processo de software e serviços. O foco principal desse modelo é nas micros, pequenas e médias empresas do mercado de software, pois elas, geralmente, têm poucos recursos financeiros para implantar um processo de melhoria que tem um custo muito elevado. Outro ponto observado por eles é que o desenvolvimento de produtos de softwares no Brasil é bem alto e, portanto as empresas deveriam buscar maturidade e excelência em seus projetos de software para que assim continuem garantindo a satisfação dos clientes.

Esses fatos serviram de motivação para a Softex e seus parceiros desenvolverem um modelo que possibilitasse a redução de custos nos processos de software. Sendo assim, as empresas poderiam alcançar padrões de certificação nos softwares que desenvolvem por um preço mais acessível que o CMMI, por exemplo.

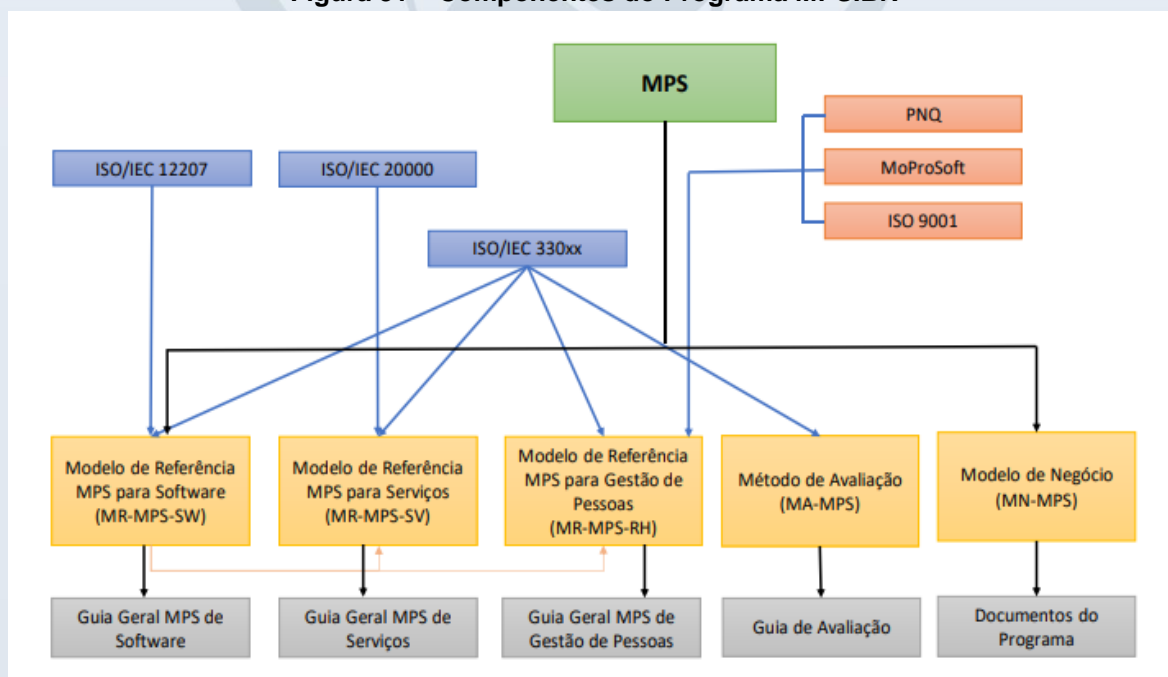
O modelo MPS está descrito através de documentos em formato de guias que são: (i) Guia Geral MPS de Software, (ii) Guia Geral MPS de serviço, (iii) Guia Geral MPS de Gestão de Pessoas e (iv) Guia de Avaliação. Ambos os guias possuem uma descrição da estrutura dos modelos, seus componentes e definições para facilitar a

compreensão e aplicação. Além disso, cada guia detalha o modelo de referência MPS/componente (como veremos no parágrafo que descreve os cinco componentes do MPS) ao qual se aplica (SOFTEX, 2020).

A figura 31 ilustra os cinco componentes do MPS.BR, que são (SOFTEX, 2020):

- Modelo de Referência MPS para Software (MR-MPS-SW). Esse modelo tem como base técnica as seguintes normas ISO: ISO/IEC/IEEE 12207 [ISO/IEC/IEEE, 2017] e a série de normas ISO/IEC 330xx e também é compatível com o CMMI-DEV. Esse modelo está escrito na Guia Geral MPS de Software.
- Modelo de Referência MPS para Serviços (MR-MPS-SV). A base técnica para esse modelo constitui-se das seguintes normas ISO: ISO/IEC 20.000 e a série de normas ISO/IEC 330xx. Esse modelo está escrito na Guia Geral MPS de Serviços.
- Modelo de Referência MPS para Gestão de Recursos Humanos (MR-MPS-RH). Já esse modelo é composto pelas seguintes normas ISO: NBR ISO 9001:2015 e a ISO/IEC 33020. Estando escrito na Guia Geral MPS de gestão de pessoas.

**Figura 31 – Componentes do Programa MPS.BR**



Fonte: Softex, 2020.

- Método de Avaliação (MA-MPS). Esse método está em conformidade com as seguintes normas ISO: ISO/IEC 33020 e ISO/IEC 33002 e ISO/IEC 33004. Esse modelo está descrito no Guia de Avaliação que possui uma descrição de todos os elementos e processos de avaliação do software baseados no modelo MPS.
- Modelo de Negócio (MN-MPS). Esse modelo apresenta as regras de negócio para: (i) implementação dos modelos MPS pelas Instituições Implementadoras (II), (ii) avaliação seguindo o MA-MPS pelas Instituições Avaliadoras (IA), (iii) organização de grupos de empresas, para implementação e avaliação de acordo com os modelos MPS, pelas Instituições Organizadoras de Grupos de Empresas (IOGE) e (iv) realização de treinamentos oficiais do MPS por meio de cursos, provas e workshops.<sup>37</sup>

**SAIBA MAIS**

Assunto: Guias no MPS.BR de 2020

Disponível no link: <<https://softex.br/mpsbr/guias/#toggle-id-3>>.

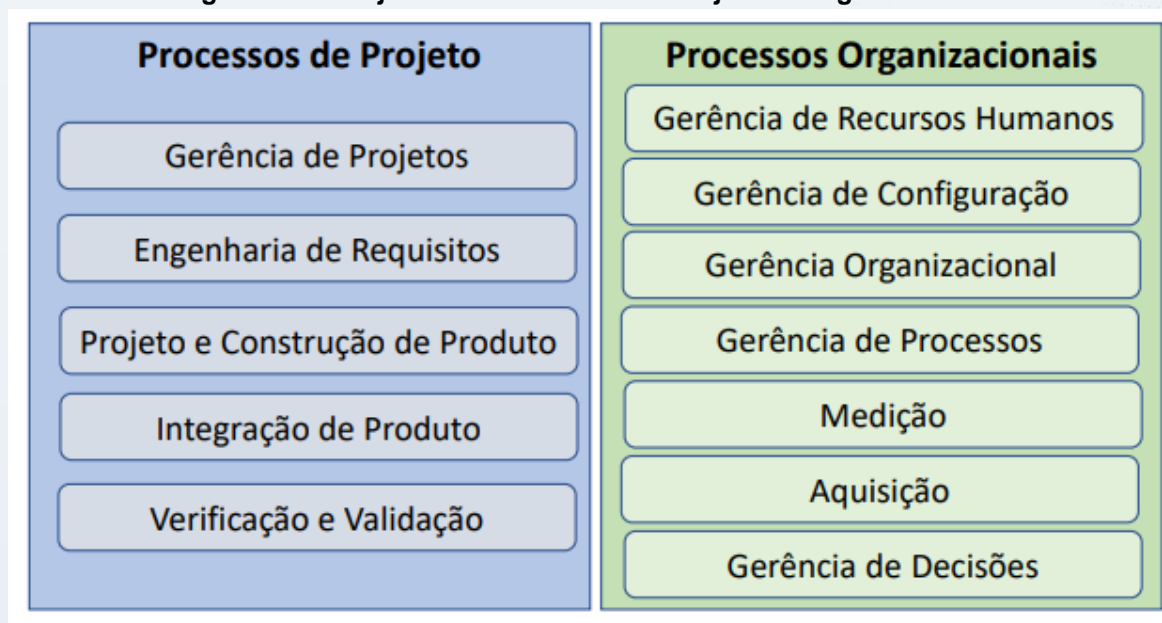
**3.2.1 O Modelo de Referência MPS para Software**

O Modelo de Referência MPS para Software (MR-MPS-SW) define níveis de maturidade que são uma combinação entre a sua capacidade e processos, os quais estão divididos em dois conjuntos que são definidos a seguir e ilustrados na figura 32 (SOFTEX, 2020):

<sup>37</sup> Um resumo executivo das regras de negócio está disponível em <http://softex.br/mpsbr/>.



**Figura 32 – Conjunto de Processos de Projetos e Organizacionais**



Fonte: Softex, 2020.

- Processos de Projetos refere-se aqueles processos que são executados para o projeto de software. Eles podem ser de desenvolvimento de um novo produto de software, manutenção ou evolução do sistema.
- Processos organizacionais tratam-se dos processos estabelecidos para conceder os recursos essenciais para que o projeto ou serviço atenda às expectativas e necessidades dos stakeholders da empresa.

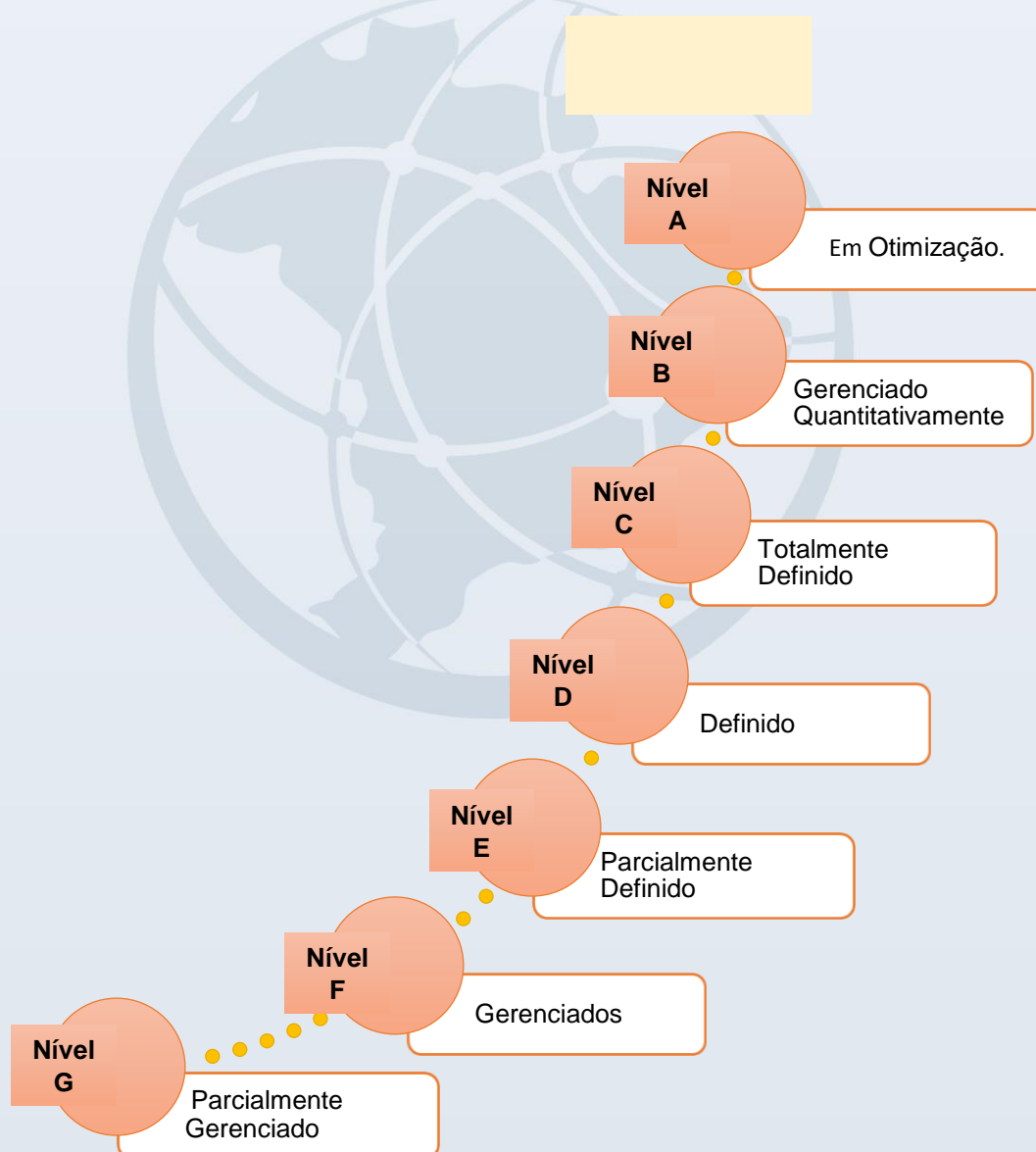
A figura 33 apresenta os sete níveis de maturidade que vão de A (Melhor) até o G (Pior). Tal separação facilita que a empresa planeje o seu desempenho futuro e saiba em qual nível de maturidade ela está (SOFTEX, 2020). Os resultados dos processos são acumulativos, dessa forma, os resultados que aparecem no nível G estarão presentes no nível F e assim sucessivamente até o Nível A, claro que se este for o foco da empresa.

Os resultados esperados, atividades e tarefas indispensáveis para atender ao propósito das organizações estão definidas detalhadamente no Guia do MPS.BR<sup>38</sup>. Nessa apostila, os níveis de maturidade serão abordados de forma resumida devido a sua extensibilidade.

<sup>38</sup> [https://softex.br/download/mps-br\\_guia\\_geral\\_software\\_2020\\_versao\\_maio\\_2020/](https://softex.br/download/mps-br_guia_geral_software_2020_versao_maio_2020/)

O nível G – *Parcialmente Gerenciado* - trata-se de um nível de maturidade bem inferior. Nesse nível, ocorre uma gerência de projetos e de requisitos de software. Entretanto, na gerência de projetos, existem algumas tarefas fundamentais que carecem ser realizadas em qualquer projeto de software, por exemplo, a identificação e monitoramento de atividades referentes ao projeto. Já na gerência de requisitos de softwares, observa-se que a empresa já possui uma boa habilidade de lidar com as modificações que os requisitos de software enfrentam no decorrer do desenvolvimento de um projeto de software (GASPAROTTO, 2013).

**Figura 33 - Evolução dos Processos nos Níveis de Maturidade do MR-MPS-SW.**



**Fonte: Elaborado pela autora, 2020.**

No nível F – Gerenciado - pode-se considerar que a empresa já tem uma maturidade um pouco mais alta, em que já há uma garantia da qualidade do produto, como o controle nas aquisições de produtos e serviços, gerência de configuração e medição do software. Posteriormente, tem-se o nível E - Parcialmente Definido - que é o responsável por garantir uma melhor organização nos processos. Esse nível traz adaptação do processo para gerência de projeto, definição do processo organizacional, processo de treinamento em toda a empresa (GASPAROTTO, 2013).

Já o nível D – Largamente Definido - consiste em um nível de maturidade intermediário que traz algumas ideias de desenvolvimento de requisitos de software, como a utilização de técnicas para assegurar a qualidade do software pela validação e verificação. Esse nível possibilita que a empresa seja capaz de descobrir os requisitos junto com o cliente. Dessa forma, não haverá a necessidade de realizar a confirmação que esses requisitos foram cumpridos (GASPAROTTO, 2013).

O nível C – Definido - refere-se a alguns processos mais avançados, como: (i) gerência de riscos, que trata da diminuição dos riscos que as mudanças de pessoal podem acarretar para os projetos tentando evitar atrasos no cronograma; e (ii) análise de decisão e resolução tem a função de analisar as decisões que foram tomadas baseadas em um processo formal de avaliação. Em seguida, tem-se o nível B – Gerenciado Quantitativamente - que é o responsável por propiciar uma gerência quantitativa do projeto e calcular o desempenho do processo organizacional. A gerência quantitativa do projeto possibilita monitorar e determinar os projetos em relação a suas metas de qualidade e de desempenho o que garantirá que o software terá qualidade. Já o desempenho do processo organizacional consiste em um cálculo para que se tenha uma compreensão da qualidade dos processos que a empresa está realizando (GASPAROTTO, 2013).

E, por fim, o nível A – Em Otimização. As empresas que alcançam o nível A significa que elas possuem uma maturidade plena. Nesse nível, existem dois processos: (i) processos de análise de causas e resolução e (ii) processos de inovação na empresa. O objetivo do processo de análise de causas e resolução é identificar, e, conseqüentemente, evitar no futuro que defeitos sejam encontrados nos processos ou nos projetos realizados. Já o processo de inovação é considerado

uma parte essencial para as empresas, o fato delas estarem em busca de melhorias e evolução, as mantém consolidada no mercado (GASPAROTTO, 2013).

### SAIBA MAIS

Assunto: Certificação do MPS.BR

Disponível no link: <<http://asrconsultoria.com.br/index.php/certificacao/>>.

### 3.3 Comparação entre os modelos CMMI e MPS.BR

Os modelos de melhoria de processo CMMI e MPS.br possuem diferenças e semelhanças. Estas estão detalhadas no quadro 6.

**Quadro 6 – Diferenças e semelhanças do CMMI e MPS.BR.**

CMMI	MPS.BR
<b>O Modelo de Qualidade CMMI é reconhecido internacionalmente.</b>	O MPS.BR é mais conhecido nacionalmente e na América Latina.
<b>O modelo CMMI envolve um grande custo na Avaliação e Certificação do Modelo.</b>	No MPS.BR o custo da certificação é mais acessível.
<b>No CMMI é necessário investir tempo, geralmente para se chegar aos níveis de maturidade mais altos.</b>	No MPS.BR as avaliações são bienais.
<b>O CMMI tem foco global voltado para empresas de maior porte.</b>	MPS.BR é um modelo criado em função das médias e pequenas empresas.
<b>O CMMI possui cinco níveis de maturidade por estágio e seis na Contínua.</b>	MPS.BR possui sete níveis de maturidade, onde a implantação é mais gradual.
<b>O CMMI é aceito como maturidade para licitações.</b>	O MPS.BR é aceito como maturidade para licitações.
<b>O CMMI torna as empresas competitivas internacionalmente.</b>	O MPS.BR não torna as empresas competitivas internacionalmente.
<b>O CMMI não utiliza contrato conjunto de empresas.</b>	No MPS.BR pode acontecer contrato cooperado em grupo de empresas que queiram a Certificação.
<b>Implementação mais complexa.</b>	Implementação mais simples.

Fonte: Franciscani and Pestili (2012. p.10 e 11, *apud Oliveira, 2008*).



Dentre as principais semelhanças pode-se citar o fato de ambos terem sido criados com base nas normas ISO/IEC 12207 e ISO/IEC 15504 e também ambos são trabalhos com o desenvolvimento e gerenciamento de requisitos. Entretanto, os níveis de maturidade do CMMI servem para determinar a capacidade da organização em trabalhar em projetos complexos e grandes, enquanto o MPS.BR possibilita a organização implantar, avaliar e adquirir um reconhecimento gradual da melhoria de processo de um software. Essa prática facilita as empresas pequenas e médias implantarem esse processo de melhoria (Qualidadesw<sup>39</sup>. Acesso em: 01/07/2020).

A diferença mais aparente entre esses modelos está em relação aos níveis, conforme ilustra o quadro 7, o CMMI alterna seus níveis entre 1 e 5 no caso do CMMI por Estágio e 0 a 5 quando se trata do CMMI Contínuo. Já o MPS.BR se alterna entre os níveis G ao A, conforme observa no quadro 7 o CMMI não possui níveis correspondentes para os níveis G, E e D do MPS.BR.

**Quadro 7 - Comparação do CMMI e MPS.BR em relação aos níveis de maturidade.**

Níveis de Maturidade	CMMI	MPS.BR
0		
1	Inicial	Não é definido
		G (Parcialmente Gerenciado)
2	Gerenciado	F (Gerenciado)
		E (Parcialmente Definido)
		D (Largamente Definido)
3	Definido	C (Definido)
4	Gerenciado Quantitativamente	B (Gerenciado Quantitativamente)
5	Otimizado	A (Em Otimização)

Fonte: Qualidadesw<sup>38</sup>. Acesso em: 01/07/2020.

<sup>39</sup> Comparação entre os Modelos CMMI-DEV e MPS.BR. Disponível em: <http://qualidadesw.comunidades.net/comparacao-entre-os-modelos-cmmi-dev-e-mpsbr>. Acesso em: 01/07/2020.

### 3.4 Outras estruturas de melhoria de processo de software

Além dos dois modelos vistos anteriormente, CMMI e MPS.BR, de acordo como Pressman (2011), existem outras estruturas que forma propostas como alternativa para melhoria de processos de software que são:

- SPICE. O modelo SPICE (*Software Process Improvement and Capability dEtermination* – em português significa Melhoria de Processo de Software e Determinação de Capacidade). O SPICE propicia uma estrutura de avaliação de melhoria de processo de software compatível com a ISO 12207 e a ISO 15504:2003. A documentação do SPICE expõe a estrutura completa para a melhoria de processo de software, como um modelo de gerenciamento de processo, normas para conduzir uma avaliação e usos dos instrumentos e ferramentas de avaliação e treinamento para os avaliadores (PRESSMAN, 2011).
- Bootstrap. O Bootstrap foi criado com o objetivo de examinar um processo de software utilizando um conjunto de melhores práticas de engenharia de software para avaliar. Da mesma forma que o modelo CMMI, o Bootstrap também possui um nível de maturidade de processo por meio dos resultados de questionários que agrupam informações sobre o processo de software e projetos de software (PRESSMAN, 2011).
- PSP e TSP. Tanto o PSP (*Personal Software Process*) quanto o TSP (*Time Software Process*) também são modelos de processos de software, porém no âmbito individual, no caso do PSP e no caso do TSP específicas de equipe. Ambos enfatizam a necessidade de ininterruptamente coletar dados sobre o trabalho que está em execução e usar esses dados para criar táticas para melhorias (PRESSMAN, 2011).
- Tickit. Trata-se de um método de auditoria para avaliar se uma empresa está em conformidade com a Norma ISO 9001:2000, que é uma norma genérica que pode ser aplicada a qualquer empresa que deseja melhorar a qualidade geral dos produtos, softwares ou serviços que ela oferece (PRESSMAN, 2011).

### INDICAÇÕES BIBLIOGRÁFICAS

PAULA FILHO, Wilson de Pádua. **Engenharia de software: projetos e processos**.  
4 ed. LTC, 2019.



## REFERÊNCIAS

ASSOCIAÇÃO PARA PROMOÇÃO DA EXCELÊNCIA DO SOFTWARE BRASILEIRO – SOFTEX. MPS.BR – Guia de Avaliação: 2020, maio 2020. Disponível em: <https://softex.br/mpsbr/guias/#toggle-id-3>. Acesso em: 15/06/2020.

BACALÁ JÚNIOR, Sílvio. **Arquitetura em Camadas**. 22 Slides. Disponível em: <<http://www.facom.ufu.br/~bacala/PI/WebCamadas.pdf>>. Acesso em: 22/06/2020.

BEZERRA, Eduardo. Princípios de análise e projeto de sistemas com UML. 3 ed. Rio de Janeiro: Campus, 2015.

BOOCH, Grady; JACOBSON, Ivar; RUMBAUGH, James. **UML – guia do usuário**. 2 ed. Rio de Janeiro: Elsevier, 2005.

FRANCISCANI, Juliana de Fátima and PESTILI, Ligia Cristina. 2012. **CMMI e MPS.BR: Um Estudo Comparativo**. Disponível em: <http://www.unicerp.edu.br/images/revistascientificas/3%20-%20CMMI%20e%20MPS.BR%20Um%20Estudo%20Comparativo1.pdf>. Acesso em: 01/07/2020.

GASPAROTTO, Henrique Machado. 2014. Melhoria do Processo de Software Brasileiro. Disponível em: <https://www.devmedia.com.br/melhoria-do-processo-de-software-brasileiro/29915>. Acesso em: 16/06/2020.

GONÇALVES, Débora. 2019. **Como fazer a análise de viabilidade de projetos de aplicativos?**. Disponível em: <https://blog.cronapp.io/analise-de-viabilidade-de-projetos/>. Acesso em: 16/06/2020.

GUEDES, Marylene. 2019. TreinaWeb. **A importância da documentação para o desenvolvimento de software**. Disponível em: <https://www.treinaweb.com.br/blog/a-importancia-da-documentacao-para-desenvolvedores-de-software/>. Acesso em: 24/06/2020.

LINSTEDT, Daniel and OSLCHIMKE, Michael. 2016. **The Data Vault 2.0 Methodology**. Disponível em: <https://www.sciencedirect.com/topics/computer-science/capability-maturity-model-integration>. Acesso em: 23/06/2020.

MELO, Clarissa. 2017. Mobiliza. **Treinamento de software: 4 formas de treinar seus clientes**. Disponível em: <https://mobiliza.com.br/treinamento-de-software-4-formas-de-treinar-seus-clientes/>. Acesso em: 24/06/2020.

NEPOMUCENO, Gabriel. 2012. **Modelo Cascata, Linear ou Clássico**. Disponível em: <http://engenhariadesoftwareuesb.blogspot.com/2012/12/ffrrrrr.html>. Acesso em: 18/06/2020.



OSIS, Janis and DONINS, Uldis. 2017. **Software Designing With Unified Modeling Language Driven Approaches**. Elsevier. Disponível em: <https://www.sciencedirect.com/topics/computer-science/unified-process>. Acesso em: 18/06/2020.

OLIVEIRA, Celismar. 2020. **O que é Viabilidade**. Disponível em: <https://projetoseti.com.br/o-que-e-viabilidade/>. Acesso em: 16/06/2020.

OLIVEIRA, Camila da Silva. **Comparando CMMI x MPS.BR: As Vantagens e Desvantagens dos Modelos de Qualidade no Brasil**, 2008. Disponível em: <http://www.camilaoliveira.net/Arquivos/Comparando%20CMMi%20x%20MPS.pdf>

PRESSMAN, Roger S. MAXIM, Bruce R. **Engenharia de Software - Uma Abordagem Profissional**. 7.ed. Porto Alegre: Amgh Editora, 2011.

PRESSMAN, Roger S. MAXIM, Bruce R. **Engenharia de Software - Uma Abordagem Profissional**. 8.ed. Porto Alegre: Amgh Editora, 2016.

SOMMERVILLE, Ian. **Engenharia de Software**. 9.ed. Editora Persson, 2011.

SOMMERVILLE, Ian. **Software Engineering**. 10.ed. São Paulo: Pearson, 2015.

SOMMERVILLE, Ian. **Engenharia de Software**. Tradução de Luiz Cláudio Queiroz. 10.ed. São Paulo: Pearson Education do Brasil, 2018.

VALENTE, Marco Tulio. **Engenharia de Software Moderna: princípios e práticas para desenvolvimento de software com produtividade**. Leanpub, 2020.

VENTURA, Plínio. 2019. Entendendo o Diagrama de Atividades da UML. Disponível em: <https://www.ateomomento.com.br/uml-diagrama-de-atividades/> .Acesso em: 25/06/2020.

VERGILIO, Silvia Regina. **Unified Process (Processo Unificado)**. 33 slides. Disponível em: <<http://www.inf.ufpr.br/silvia/ES/SweES/pdf/RUPAI.pdf>>. Acessado em: 21/06/2020.

# REQUISITOS OBRIGATÓRIOS PARA CONCLUSÃO DO CURSO



Todos os cursos deverão ter duração mínima exigida, contada a partir do pagamento da primeira mensalidade e considerada a data de colação de grau do curso superior.



Ser aprovado em todas as disciplinas com nota mínima de 7 pontos.



Ter quitado todas as parcelas do curso.



Entregar todas as documentações exigidas para emissão do certificado.



Ser aprovado no TCC (Artigo ou monografia) com nota mínima de 7 pontos.



**GRUPO  
Prominas**  
EDUCAÇÃO E TECNOLOGIA

**0800 283 8380**