

# **Tópicos em Engenharia de Software**

**Texto Preliminar - Versão ECI-ES Turma 2023 02**

Geraldo Xexéo

28 de agosto de 2023 22:38

Copyright © 2023 Geraldo Xexéo.

Todos os direitos reservados.

Esta é um pré-publicação com o texto parcial em trabalho.

**Apenas os alunos da cadeira de Engenharia de Software de Engenharia de Computação e Informação, Turma de 2023/02, estão autorizados a ter uma cópia digital e uma cópia impressa deste livro.**

Contato com o autor pelo e-mail [xexeo@cos.ufrj.br](mailto:xexeo@cos.ufrj.br).

# Sumário

<b>Lista de Figuras</b>	<b>ix</b>
<b>Lista de Tabelas</b>	<b>xiii</b>
<b>1. Prefácio: A Engenharia de Software e o Mundo Real</b>	<b>3</b>
1.1. Livros são Versões . . . . .	3
1.2. Este Livro é uma Versão . . . . .	4
1.3. Níveis de Dificuldade . . . . .	4
1.3.1. O Framework Cynefin . . . . .	4
1.4. A Engenharia de Software Está Sempre Certa? . . . . .	5
<b>I. Introdução</b>	<b>7</b>
<b>2. Introdução a Engenharia de Software</b>	<b>9</b>
2.1. O que é Software . . . . .	10
2.1.1. Como cobrar? A pergunta mais comum nos cursos de ES. . . . .	11
2.2. Por que software é importante . . . . .	13
2.2.1. A Ubiquidade do Software . . . . .	14
2.3. Quais os riscos que software traz? . . . . .	16
2.4. Por que é difícil fazer software? . . . . .	17
2.4.1. O Relatório CHAOS . . . . .	19
2.5. O que é Engenharia de Software . . . . .	21
2.5.1. Uma Visão da Engenharia de Software . . . . .	22
2.6. Por que a Engenharia de Software é necessária? . . . . .	22
2.6.1. Diferenças para outras engenharias . . . . .	22
2.7. Qual o Segredo do Sucesso de um Projeto de Software . . . . .	23
2.8. O Corpo de Conhecimento da ES . . . . .	24

## Sumário

2.9.	As normas da Engenharia de Software . . . . .	25
2.10.	SEMAT . . . . .	26
2.10.1.	O Kernel Essencial . . . . .	26
2.11.	Quem é o Engenheiro de Software . . . . .	29
2.12.	Exercícios . . . . .	29
<b>3. Valor</b>		<b>31</b>
3.1.	Conceituação Genérica de Valor . . . . .	32
3.2.	Valor na Economia . . . . .	33
3.2.1.	Utilidade . . . . .	33
3.2.2.	Microeconomia Moderna e o Custo de Oportunidade . . . . .	34
3.3.	Valor em Marketing . . . . .	35
3.3.1.	Os Elementos do Valor - B2C . . . . .	36
3.3.2.	Os Elementos do Valor - B2B . . . . .	37
3.3.3.	Outras Técnicas . . . . .	40
3.4.	Priorização Baseada em Valor para o Cliente . . . . .	41
3.4.1.	Técnicas simples de ordenação . . . . .	41
3.4.2.	Método MoSCoW . . . . .	41
3.4.3.	Análise de Kano . . . . .	42
3.4.4.	Prioridade e Consenso . . . . .	43
3.4.5.	Usando mais de uma dimensão . . . . .	45
3.4.6.	Prioridade e Dependências . . . . .	47
3.5.	Valor Financeiro de Projetos . . . . .	48
3.5.1.	Custo Total de Propriedade . . . . .	49
3.5.2.	Retorno do Investimento . . . . .	50
3.5.3.	Valor Presente Líquido . . . . .	51
3.5.4.	Taxa Interna de Retorno . . . . .	52
3.6.	Valor em Software . . . . .	53
3.6.1.	Triângulo do Valor . . . . .	54
3.6.2.	Valor em Métodos Ágeis . . . . .	54
3.7.	Conclusão . . . . .	55
3.8.	Exercícios . . . . .	55
<b>4. 5W2H</b>		<b>59</b>
4.1.	As Perguntas Certas . . . . .	60
4.2.	Usos do 5W2H . . . . .	61
4.2.1.	O <i>Framework de Zachman</i> . . . . .	62
4.3.	Onde Está o Valor? . . . . .	63
4.4.	Várias Perguntas para cada Palavra . . . . .	64
4.4.1.	Perguntas sobre o que . . . . .	64
4.4.2.	Perguntas sobre quem . . . . .	64
4.4.3.	Perguntas sobre quando . . . . .	65
4.4.4.	Perguntas sobre onde . . . . .	65
4.4.5.	Perguntas sobre como . . . . .	66

4.4.6. Perguntas sobre por que . . . . .	66
4.4.7. Perguntas sobre quanto . . . . .	66
4.5. Exercícios . . . . .	67
<b>5. Partes Interessadas</b>	<b>69</b>
5.1. O Que São Partes Interessadas . . . . .	71
5.1.1. Usuários . . . . .	73
5.1.2. Perspectivas dos Usuários . . . . .	73
5.2. Partes Interessadas e Valor . . . . .	75
5.3. Gerência das Partes Interessadas . . . . .	75
5.4. A abordagem SEMAT para Partes Interessadas . . . . .	76
5.5. Caracterização da Parte Interessada . . . . .	76
5.5.1. Interesses e Objetivos . . . . .	77
5.6. Classificação das Partes Interessadas . . . . .	79
5.6.1. Análise das Partes Interessadas . . . . .	81
5.7. Representação das Partes Interessadas . . . . .	83
5.8. Engajamento das Partes Interessadas . . . . .	83
5.9. Matriz RACI . . . . .	83
5.10. Conclusão . . . . .	85
5.11. Exercícios . . . . .	85
<b>6. Qualidade</b>	<b>87</b>
6.1. Tipos de Qualidade . . . . .	90
6.2. Dimensões da Qualidade . . . . .	90
6.3. Qualidade é Sempre por um Ponto de Vista . . . . .	95
6.4. Qualidade de Produto e Qualidade de Processo . . . . .	96
6.5. Falhas e Defeitos . . . . .	96
6.6. Quanto Custa a Qualidade . . . . .	97
6.7. Gestão da Qualidade . . . . .	99
6.8. Exercícios . . . . .	99
<b>II. Os Clientes</b>	<b>101</b>
<b>7. Oportunidades e Problemas</b>	<b>103</b>
7.1. Oportunidades . . . . .	104
7.2. Problemas . . . . .	105
7.3. Tipos de Problemas . . . . .	106
7.4. Identificando Problemas . . . . .	106
7.4.1. A Análise de Pareto . . . . .	107
7.4.2. Calculando a Quantidade . . . . .	108
7.4.3. Usando o Impacto dos Problemas . . . . .	109
7.4.4. Buscando Causas . . . . .	111
7.4.5. Organização do Diagrama Espinha de Peixe . . . . .	114

## Sumário

7.5.	Caracterização do Problema . . . . .	115
7.6.	Oportunidade e 5W2H . . . . .	116
7.6.1.	Buscando Oportunidades com as Partes Interessadas . . . . .	116
7.7.	Propondo Oportunidades ao Cliente . . . . .	117
7.8.	Oportunidade e Valor . . . . .	118
7.9.	Exercícios . . . . .	118
<b>8. Requisitos</b>		<b>121</b>
8.1.	Definição Formal . . . . .	122
8.1.1.	Exemplos de Requisitos . . . . .	123
8.2.	Requisitos x Benefícios . . . . .	126
8.3.	Tipos de Requisitos . . . . .	126
8.3.1.	Requisitos Funcionais . . . . .	126
8.3.2.	Requisitos não funcionais . . . . .	126
8.3.3.	Restrições . . . . .	128
8.3.4.	Outros tipos de requisitos . . . . .	128
8.4.	O Papel dos Requisitos no Projeto . . . . .	128
8.5.	Engenharia de Requisitos . . . . .	129
8.6.	Elicitação de Requisitos . . . . .	129
8.6.1.	Processo de Elicitação de Requisitos . . . . .	131
8.7.	Níveis de Abstração dos Requisitos . . . . .	134
8.8.	O Risco de Requisitos Falsos . . . . .	134
8.9.	Características de um Bom Requisito . . . . .	135
8.9.1.	Efeitos de Requisitos com Problemas . . . . .	136
8.10.	Atributos dos Requisitos . . . . .	137
8.10.1.	Volere e o <i>snowcard</i> . . . . .	137
8.11.	Priorizando Requisitos . . . . .	139
8.12.	Requisitos Mudam com o Tempo . . . . .	140
8.13.	Requisitos e Necessidades . . . . .	141
8.14.	Requisitos de Acordo com Normas IEEE . . . . .	142
8.14.1.	Requisitos Funcionais na Norma IEEE . . . . .	143
8.14.2.	O Documento de Especificação de Requisitos . . . . .	144
8.15.	A Visão do Novo Sistema . . . . .	145
8.16.	Descrevendo Requisitos . . . . .	145
8.16.1.	Exemplos de Requisitos . . . . .	146
8.17.	Exercício . . . . .	147
<b>III. Gestão de Software</b>		<b>149</b>
<b>9. O Que é um Projeto</b>		<b>151</b>
9.1.	Recursos e Entregas . . . . .	152
9.2.	Partes Interessadas . . . . .	153

9.3.	Sucesso, Fracasso e Riscos . . . . .	153
9.3.1.	A Incerteza dos Projetos e os Riscos . . . . .	153
9.3.2.	Benefícios da Gerência de Projeto . . . . .	154
9.4.	Pontos Críticos ou Pontos Chave de Sucesso . . . . .	154
9.5.	Conclusão . . . . .	155
<b>10. Gestão de Projetos de Software</b>		<b>157</b>
10.1.	Técnicas Tradicionais de Gestão de Projeto . . . . .	158
10.1.1.	Estrutura Analítica de Projeto . . . . .	158
10.1.2.	Controlgrama de Gantt . . . . .	159
10.1.3.	Redes PERT/CPM . . . . .	160
10.1.4.	Algoritmo para cálculo das datas em uma rede PERT/CPM . . . . .	164
10.2.	Uso das Técnicas Tradicionais . . . . .	166
10.3.	Técnicas Ágeis de Gestão de Projeto . . . . .	166
10.3.1.	Kanban . . . . .	166
<b>IV. Especificação</b>		<b>167</b>
<b>11. Casos de Uso</b>		<b>169</b>
11.1.	Conceituação de Caso de Uso . . . . .	172
11.2.	Conceitos Importantes nos Casos de Uso . . . . .	173
11.2.1.	Ator . . . . .	173
11.2.2.	Objetivos . . . . .	174
11.2.3.	Cenários . . . . .	175
11.2.4.	As Alternativas . . . . .	179
11.2.5.	Casos de Uso em UML . . . . .	180
11.3.	A Narrativa do Caso de Uso . . . . .	180
11.4.	O Nível de Abstração . . . . .	181
11.5.	O Escopo do Caso de Uso . . . . .	182
11.6.	Casos de Uso Especiais . . . . .	183
11.7.	Partes do Caso de Uso . . . . .	183
11.8.	Exemplo de Um Caso de Uso . . . . .	184
11.9.	Levantando Casos de Usos . . . . .	185
11.10.	Diagramas de Caso de Uso . . . . .	188
11.10.1.	Especificando o <i>Subject</i> . . . . .	189
11.10.2.	Generalização entre Atores . . . . .	190
11.10.3.	Relações Entre Casos de Uso . . . . .	191
11.10.4.	Usando Cardinalidades em Casos de Uso . . . . .	194
11.11.	O Que o Diagrama de Caso de Uso Não Informa . . . . .	195
11.12.	Exercícios . . . . .	195
<b>12. Histórias do Usuário</b>		<b>197</b>
12.1.	Conceituação de Histórias do Usuário . . . . .	199

## Sumário

12.2.	Um <i>Template</i> Padrão para Histórias de Usuário . . . . .	200
12.2.1.	Outros formatos possíveis . . . . .	201
12.3.	INVEST . . . . .	202
12.3.1.	Independente . . . . .	202
12.3.2.	Negociável . . . . .	204
12.3.3.	Com Valor . . . . .	204
12.3.4.	Estimável . . . . .	204
12.3.5.	Pequena . . . . .	205
12.3.6.	Testável . . . . .	206
12.4.	Estados de Uma História do Usuário . . . . .	206
12.5.	Exercícios . . . . .	206
<b>V. Desenvolvimento</b>		<b>207</b>
<b>13. Testes</b>		<b>211</b>
13.1.	Terminologia . . . . .	212
13.2.	Introdução aos Conceitos de Teste . . . . .	212
<b>VI. Métricas de Software</b>		<b>215</b>
<b>14. Medidas e Estimativas do Tamanho do Software</b>		<b>217</b>
14.1.	Esforço . . . . .	219
14.2.	Algumas Medidas Conceitualmente Simples . . . . .	220
14.2.1.	A Medida Mais Simples: Linhas de Código . . . . .	220
14.2.2.	Pontos de História . . . . .	222
14.2.3.	Tamanhos de Camisa . . . . .	223
14.3.	Visão Geral dos Métodos de Estimativa . . . . .	224
14.4.	Métodos de Estimativa Baseado em Opinião . . . . .	225
14.4.1.	O Método de Estimativa do PERT . . . . .	225
14.4.2.	O Método de Delphi . . . . .	225
14.4.3.	O Planning Poker . . . . .	225
<b>15. Análise de Pontos de Função</b>		<b>227</b>
15.1.	Visão Geral de Pontos de Função . . . . .	228
15.2.	Procedimento de Contagem . . . . .	228
15.3.	Funções Transacionais . . . . .	229
15.4.	Funções de Dados . . . . .	230
15.5.	Identificando Funções de Negócio . . . . .	230
15.5.1.	Identificando Saídas . . . . .	231
15.5.2.	Identificando Consultas . . . . .	231
15.6.	Identificando Entradas . . . . .	231
15.6.1.	Entendendo as Lógicas de Processamento . . . . .	232

15.7.	Identificando Arquivos Internos . . . . .	233
15.7.1.	Identificando Arquivos Externos . . . . .	234
15.7.2.	Identificando Itens de Dados (DETs) . . . . .	234
15.8.	As Perguntas . . . . .	236
15.9.	Cálculo dos Pfs Finais . . . . .	236
15.10.	Conclusão . . . . .	237
<b>16.</b>	<b>Uma Visão Reduzida do Modelo COCOMO II</b>	<b>239</b>
16.1.	Dois Modelos em Um . . . . .	240
16.2.	Esforço em Função do Tamanho . . . . .	240
16.3.	Existe uma Relação Entre Esforço e Tempo . . . . .	241
16.3.1.	Cálculo dos Fatores de Escala . . . . .	242



# **Lista de Figuras**

2.1.	Um software executando na web e parte do seu código. . . . .	11
2.2.	Sala de controle de uma usina nuclear, feita originalmente com tecnologia analógica e modernizada com computadores e software . . . . .	14
2.3.	Esquema da virtualização do hardware. . . . .	15
2.4.	Tamanho de código de alguns projetos conhecidos. . . . .	19
2.5.	Curva de falha do hardware . . . . .	20
2.6.	Curva de falha do software . . . . .	21
2.7.	Pirâmide que explica a construção de conceitos na Engenharia de Software. . . . .	22
2.8.	Mind map para as áreas de conhecimento da Engenharia de Software: Fonte:(IEEE Computer Society, 2014a) . . . . .	25
2.9.	O Kernel Essencial do SEMAT. Fonte:(Jacobson, Ng et al., 2013) . .	27
2.10.	Os Espaços de Atividades no Kernel do SEMAT. Fonte:(Jacobson, Ng et al., 2013) . . . . .	27
2.11.	As Competências no Kernel do SEMAT. Fonte:(Jacobson, Ng et al., 2013) . . . . .	28
3.1.	Elementos do Valor B2C. Fonte:Almquist, Senior e Bloch, 2016 . . .	37
3.2.	Hierarquia das Necessidades de Maslow. Fonte: Wikipedia Commons por Felipe Sanchez (CC-BY-SA 3.0) e J. Finkelstein (GFDL) . . . . .	38
3.3.	Elementos do Valor B2B. Fonte: Almquist, Cleghorn e Sherer, 2018 .	39
3.4.	. . . . .	44
3.5.	Gráfico com quadrantes para comparar prioridades usando benefício e custo. . . . .	46
3.6.	Gráfico com quadrantes, determinados arbitrariamente pelo usuário, para os itens de projeto listados na Tabela 3.2. As setas mostram uma possível sequência de prioridade. Os itens C e D foram considerados desnecessários ao projeto. . . . .	47

## *Lista de Figuras*

3.7.	Cálculo do valor futuro de um investimento de R\$10.000,00 por mês com uma taxa de juros de 1% . . . . .	49
3.8.	Gráfico do TCO de impressoras em função de da quantidade de páginas impressas. Percebe-se que após aproximadamente 3500 páginas é melhor comprar uma impressora a laser mais cara, caso esse fosse o único fator de análise. . . . .	51
3.9.	O triângulo do valor para a maioria dos projetos. . . . .	54
4.1.	O Framework de Zachman fornece uma visão da arquitetura empresarial a partir de 6 dimensões definidas pelo 5WH. Reprinted with permission by John A. Zachman, Zachman International. . . . .	63
5.1.	O Kernel Essencial do SEMAT. Fonte: (Jacobson, Ng et al., 2013) . .	70
5.2.	O impacto das partes interessadas. . . . .	71
5.3.	3 tipos comuns de usuários dos sistemas. Fonte:próprio autor . . . . .	74
5.4.	Quadro para o tratamento das partes interessadas. Fonte: (PMI, 2017)	80
5.5.	Classificação das partes interessadas. Fonte: (Mitchell, Agle e Wood, 1997) . . . . .	82
6.1.	Mapa Mental da organização da área do conhecimento Qualidade de Software no SWEBOK 3.0. Fonte:(IEEE Computer Society, 2014a) .	89
6.2.	A Qualidade é a pedra basilar da Engenharia de Software. Fonte:(Presman e Maxim, 2014) . . . . .	89
6.3.	Modelo de Qualidade de McCall. Fonte:(McCall, Richards e Walters, 1977) . . . . .	92
6.4.	O Modelo Fuzzy de Qualidade de Software. Fonte:(Belchior, 1997) .	93
6.5.	Modelo de Qualidade de Produto de Software da norma ISO 25010. Fonte:(ISO, 2011) . . . . .	93
6.6.	Modelo de Qualidade em uso de Software da norma ISO 25010. Fonte:(ISO, 2011) . . . . .	94
6.7.	Diferentes pontos de vista implicam em diferentes avaliações de qualidade. Fonte: próprio autor . . . . .	95
7.1.	Gráfico com a quantidade de problemas. . . . .	109
7.2.	Pareto com custo . . . . .	112
7.3.	Um diagrama de Espinha de Peixe . . . . .	113
7.4.	Um diagrama de Espinha de Peixe resumido, construído com uma ferramenta de <i>mind map</i> . . . . .	113
7.5.	Um diagrama de Espinha de Peixe expandido, construído com uma ferramenta de <i>mind map</i> . . . . .	114
7.6.	Um Diagrama de Causa Raiz vertical, onde a causa raiz está marcada.	115
8.1.	Um cartão Volere. . . . .	124
8.2.	Exemplo de um diagrama de casos de uso . . . . .	125
8.3.	Exemplo de um mapa mental. . . . .	125

8.4.	Pequena taxonomia de requisitos não funcionais. Fonte: (Sommerville, 2015) . . . . .	127
8.5.	O papel dos requisitos no desenvolvimento do sistema. Fonte:(J. Robertson e S. Robertson, 2006) . . . . .	129
8.6.	A elicitação de requisitos é um processo interativo onde, por aproximações sucessivas, o desenvolvedor consegue a aprovação de uma especificação de requisitos. . . . .	132
8.7.	O processo de elicitação de requisitos, adaptado de Christel e Kand (1992). . . . .	132
8.8.	Cada requisito se posicionará em algum ponto desse gráfico, de acordo com as notas de avaliação. . . . .	139
10.1.	EAP em formato texto, com o mesmo significado da Figura 10.2. Fonte: do autor . . . . .	160
10.2.	Diagrama com modelo EAP equivalente ao da Figura 10.1. Fonte: do autor . . . . .	161
10.3.	Controlgrama de Gantt feito com o software Project Libre. Fonte: do autor . . . . .	162
10.4.	Modelo PERT original para a Tabela 10.1 . . . . .	163
10.5.	Um diagrama de rede construído com o software MSProject, software altamente configurável. Vemos atividades completas, em andamento. Também é possível notar as atividades no caminho crítico. Fonte: do autor . . . . .	164
10.6.	Diagrama de rede para a Tabela 10.1, feito no software Project Libre. Fonte: o autor . . . . .	164
11.1.	Exemplo de cenário principal de caso de uso. . . . .	170
11.2.	Exemplo de um caso de uso. . . . .	171
11.3.	Representação de um ator em um diagrama de caso de uso. . . . .	174
11.4.	Representação de dois atores identificados. . . . .	174
11.5.	Exemplo de cenário principal do caso de uso <b>sacar dinheiro</b> . . . . .	176
11.6.	Representação abstrata de um cenário principal e dos fluxos alternativos	178
11.7.	Representação abstrata de cada alternativa investigada nesse caso. . . . .	178
11.8.	Exemplo de cenário principal do caso de uso <b>atribuir notas</b> . . . . .	179
11.9.	Exemplo de cenário principal do caso de uso <b>emitir boletim</b> . . . . .	187
11.10.	Os elementos básicos de um diagrama de caso de uso são: atores, casos de uso e associações. Fonte: do autor . . . . .	189
11.11.	Indicando o sistema ao qual os casos de uso se referem. Fonte:do autor.	190
11.12.	Nesse diagrama de casos de uso, o gerente é um especialização de funcionário, logo pode fazer tudo que o funcionário faz, mas alguns casos de uso que são exclusivos dele. Fonte: do autor . . . . .	191
11.13.	Um <i>Profissional de Saúde</i> é a generalização de Enfermeira, Médico ou Auxiliar de Enfermagem, mas nenhum ator é um profissional de saúde se não for de um dos outros tipos. Fonte: do Autor. . . . .	191

*Lista de Figuras*

11.14.	Representações das associações entre casos de uso . . . . .	192
11.15.	Exemplo da associação de inclusão entre casos de uso . . . . .	193
11.16.	Um exemplo da associação de extensão entre casos de uso . . . . .	193
11.17.	Exemplo da associação de herança entre casos de uso . . . . .	194
11.18.	Exemplo do uso de cardinalidades em casos de uso. Fonte: do autor .	195
12.1.	Exemplo de uma história do usuário em um cartão. Fonte: do autor.	199
12.2.	Partes encontradas em um cartão de história do usuário, segundo um modelo original da Connextra (Patton e Economy, 2014a).	200
14.1.	Ilustração da sequência do Planning Poker com duas rodadas . . . . .	226
15.1.	O processo geral de contagem de Pontos de Função. Fonte: do autor .	229
15.2.	Planilha de registro para ajudar a contar os Pontos de Função . . . .	237
16.1.	Efeitos das constantes A e B na relação entre tamanho e esforço . . . .	241

# **Lista de Tabelas**

2.1. Resolução de projetos de software por tamanho e por método de desenvolvimento, para mais de 10.000 projetos. Fonte:(The Standish Group, 2015) . . . . .	20
2.2. Fatores de Sucesso de um Projeto de Software . . . . .	24
3.1. Resolvendo a classe de Kano por meio do cruzamento de duas perguntas (Moorman, 2012) . . . . .	43
3.2. Benefício e esforço para itens de um project backlog imaginário. . . . .	46
3.3. Valores usados para calcular o TCO de uma impressora. . . . .	50
3.4. Tabela de cálculo do Valor Presente Líquido, considerando saldos diferentes em cada período. Foi usada uma taxa de 5% a cada período. A fórmula para cada Valor Presente é $Caixa_i / (1 + Taxa)^{\text{Período}}$ . . . . .	52
4.1. Perguntas e respostas a partir da notícia. . . . .	61
5.1. Exemplo de formulário para registro do interesse e objetivo da parte interessada. . . . .	78
5.2. Exemplo de matriz de cruzamento para registro do interesse e objetivo da parte interessada. . . . .	79
5.3. Matriz de Engajamento: C significa comportamento corrente, D o comportamento desejado. . . . .	80
5.4. Exemplo de Matriz RACI . . . . .	84
6.1. Custos típicos de software por tamanho e nível de qualidade. Fonte:(C. Jones e Bonsignour, 2012) . . . . .	98
7.1. Uma contagem de tipos de erros para uso da técnica de Pareto . . . . .	108
7.2. Custo médio relativo dos problemas . . . . .	110
7.3. Custo total relativo . . . . .	111

## *Lista de Tabelas*

8.1. Adequação das formas de elicitação propostas por K. E. Wiegers e Beatty (2013) e alguns tipos de projeto. . . . .	130
8.2. Processo de Elicitação de Requisitos segundo Christel e Kand (1992). . . . .	133
10.1. Tabela de tarefas de um projeto . . . . .	163
10.2. Tabela de tarefas de um projeto, com as datas de início e término mais cedo . . . . .	165
10.3. Tabela de tarefas de um projeto, com as datas de início e término mais cedo e mais tarde. . . . .	165
11.1. Verbos comumente usados para nomear casos de uso . . . . .	175
11.2. Uma forma alternativa de representar os cenários possíveis a partir de um cenário principal e as alternativas associadas . . . . .	179
11.3. Ícones que representam o nível de abstração dos casos de uso. Fonte:(Cockburn, 2000) . . . . .	181
11.4. Ícones que representam o nível de escopo dos casos de uso. Fonte:(Cockburn, 2000) . . . . .	182
11.6. Associações que podem ocorrer entre atores e casos de uso. . . . .	189
15.1. Tabela de apoio a determinação do tipo de função transacional. P=possível, N=Não permitida, O=obrigatório, O <sup>*</sup> = obrigatório que pelo menos uma das condições aconteça. . . . .	233
15.2. Cálculo da complexidade das saídas externas. . . . .	235
15.3. Cálculo da complexidade das entradas externas. . . . .	235
15.4. Cálculo da complexidade das consultas externas. . . . .	235
15.5. Cálculo da complexidade dos arquivos lógicos internos. . . . .	235
15.6. Cálculo da complexidade das interfaces lógicas externas. . . . .	235
16.1. Conversão de Pontos de Função não ajustados para linhas de código. Fonte: (B. Boehm, Abts e Brad Clark, 1997) . . . . .	240
16.2. Valores das constantes no modelo COCOMO II padrão . . . . .	242

# **Explicação**

Esse livro é escrito simultaneamente com outro, que trata de Análise de Sistemas. Eles possuem capítulos comuns.



# 1

## Prefácio: A Engenharia de Software e o Mundo Real

As a rule, software systems do not work well until they have been used, and have failed repeatedly, in real applications

(*David Parnas*)

### Conteúdo

---

1.1.	Livros são Versões . . . . .	3
1.2.	Este Livro é uma Versão . . . . .	4
1.3.	Níveis de Dificuldade . . . . .	4
1.4.	A Engenharia de Software Está Sempre Certa? . . . . .	5

### 1.1. Livros são Versões

Todo livro é uma visão abstrata de algo, é uma teoria ou mesmo uma hipótese. Não existe um livro no mundo com verdades absolutas sobre o mundo real, ou fatos absolutamente verdadeiros, porque para escrever devemos escolher os detalhes, adotar perspectivas, criar modelos, etc. Livros são versões. Por exemplo, mesmo uma descrição bem detalhada de como foi um processo de desenvolvimento de software não vai saber o que as pessoas estavam sentindo, como estavam motivadas, quanto estavam realmente

## 1. Prefácio: A Engenharia de Software e o Mundo Real

felizes com seu trabalho, se pegaram um engarrafamento antes de chegar na reunião importante, etc.

### 1.2. Este Livro é uma Versão

Em especial, a Engenharia de Software existe de muitas formas. Este capítulo chama a atenção do leitor para que ele leia não só o livro, mas o analise em um nível *meta*, como é hábito falar no caso de modelos. O leitor precisa estar consciente que toda Engenharia de Software aqui descrita é primeiro uma teoria, e depois uma simplificação.

Existe uma piada em que o professor, na sala de aula ensina um problema muito simples, algo como  $x = 4 * 5 + 3$ , na prova pede algo mais complicado, algo como  $x = \sqrt{\left(\frac{94+32}{13^5}\right)}$ , e no mundo real encontramos algo ainda mais complicado, que seja  $\sum_1^n \frac{n^p}{p^n}$ . É verdade que, nos livros, começamos com os exemplos simples, porque precisamos que sejam entendidas, e o que é muito complexo, não pode ser entendido.

### 1.3. Níveis de Dificuldade

Existem várias formas de discutir a dificuldade de problemas.

#### 1.3.1. O Framework Cynefin

O **Framework Cynefin** fornece cinco contextos, ou domínios, para a tomada de decisão:

1. claro (*clear*), composto pelos *known knowns*, isto é os “conhecidos conhecidos”, onde a situação é estável e a relação causa e efeito conhecida;
2. complicado, composto pelos *known unknowns*;
3. complexo, os *unknown unknowns*;
4. caótico, e
5. confuso, não há clareza em que domínio se encontra.

<sup>1</sup>

Outra classificação fala de problemas “domados” (*tame*), para os quais existem soluções conhecidas, problemas críticos, que causam crises e precisam de respostas imediatas,

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Cynefin\\_framework](https://en.wikipedia.org/wiki/Cynefin_framework)

e problemas malditos (*wicked*), que não tem solução e muitas vezes não podem ser resolvidos, mas apenas ter o seu estado alterado.<sup>2</sup> <sup>3</sup> <sup>4</sup> <sup>5</sup>

## 1.4. A Engenharia de Software Está Sempre Certa?

O que chamamos de Engenharia de Software é formado por um corpo de conhecimento que foi criado de muitas formas. Algumas coisas foram inventadas e postas em prática, outras foram percebidas ou encontradas em projetos de sucesso, outras partiram de raciocínios e considerações sobre a prática, etc. Elas vieram de pesquisadores e profissionais do mercado, que algumas vezes possuem outros interesses além de contribuir para esse corpo de conhecimento, como vender produtos, escrever artigos, ser promovidos, etc. Algumas vezes essas pessoas têm opiniões discordantes.

Por exemplo, é provável que a polêmica entre “documentadores” e “programadores”, ou seja, aqueles que defendem projetos de software fortemente documentados, ou passando por processos extremamente rígidos, e aqueles que defendem processos ágeis fortemente focados na entrega de programas executando o que o usuário deseja, nunca termine, até porque parte dessa discussão está ligada aos contextos onde o software é criado.

A verdade é que a Engenharia de Software, como um todo, não é uma Ciência Exata, do tipo em que se pode provar com certeza que algo é verdade, mesmo que com uma margem de erro. Apesar de ser uma Engenharia, porque cria artefatos, e parte dela ser exata, grande parte dela é ligada ao trabalho das pessoas, e seus erros.

Vamos analisar um caso específico, de uma ideia que faz bastante sucesso na Engenharia de Software, e acabou se tornando quase que universalmente aceitos como verdades absolutas, os Padrões de Projeto de software propostos pela gang dos quatro.

Esses padrões de projeto (*design patterns*), como *Singleton*, *Facade* ou *Observer*. são formas de organizar programas de computador que precisam de certas funcionalidades. Por exemplo, o padrão *Observer* mostra uma solução para que vários objetos, observadores, possam estar cientes do valor atual de um outro objeto, observado, basicamente por meio do que chamamos de um *call back*.

Esses padrões de projeto são realmente as melhores soluções? Como podemos saber isso?

A maioria dos leitores **aceita** o discurso dos autores, que nos garantem que certas práticas são melhores que outras, dando algumas explicações, porque ela está “vestida” com a “autoridade” de um livro, e referenciada como boa opinião por vários outros profissionais.

---

<sup>2</sup><https://www.leadershipcentre.org.uk/artofchangemaking/theory/critical-tame-and-wicked-problems/>

<sup>3</sup>[http://www.drbird.com/problems\\_and\\_solutions/three\\_types\\_of\\_problems.html](http://www.drbird.com/problems_and_solutions/three_types_of_problems.html)

<sup>4</sup>[https://en.wikipedia.org/wiki/Wicked\\_problem](https://en.wikipedia.org/wiki/Wicked_problem)

<sup>5</sup><https://pubsonline.informs.org/doi/epdf/10.1287/mnsc.14.4.B141>

## *1. Prefácio: A Engenharia de Software e o Mundo Real*

Poucos leitores, portanto, avaliam cada padrão de projeto isoladamente, pesam vantagens e desvantagens, e decidem se querem usá-los ou não. De certa forma, aceitam o que está escrito em um livro técnico como um religioso aceita o que está escrito em um livro sagrado. Esse fenômeno é bem conhecido pelos estudiosos da Ciência e Tecnologia.

Sugiro que os leitores deste livro o encarem de forma crítica, principalmente com o passar do tempo. Alguma informação que está aqui tem seu valor intrínseco, outras um valor histórico, e ainda outras tem seu valor momentâneo e passageiro, devido ao avanço das tecnologias.

# **Parte I.**

## **Introdução**



# 2

## Introdução a Engenharia de Software

Most software today is very much like an Egyptian pyramid with millions of bricks piled on top of each other, with no structural integrity, but just done by brute force and thousands of slaves.

---

(Alan Kay)

### Conteúdo

---

2.1.	O que é Software . . . . .	10
2.2.	Por que software é importante . . . . .	13
2.3.	Quais os riscos que software traz? . . . . .	16
2.4.	Por que é difícil fazer software? . . . . .	17
2.5.	O que é Engenharia de Software . . . . .	21
2.6.	Por que a Engenharia de Software é necessária? . . . . .	22
2.7.	Qual o Segredo do Sucesso de um Projeto de Software . . . . .	23
2.8.	O Corpo de Conhecimento da ES . . . . .	24
2.9.	As normas da Engenharia de Software . . . . .	25
2.10.	SEMAT . . . . .	26
2.11.	Quem é o Engenheiro de Software . . . . .	29

## 2. Introdução a Engenharia de Software

2.12. Exercícios . . . . .	29
----------------------------	----

Este capítulo faz uma pequena introdução a área de Engenharia de Software.

### 2.1. O que é Software

**Software** é uma palavra da língua inglesa usada em contraposição a palavra **hardware**, que descreve máquinas e aparatos físicos de forma geral. Assim, um computador seria dividido em uma parte física, composta de circuitos eletrônicos, e uma parte não-física, composta do programa que o faz funcionar.

#### Quem inventou a palavra software?

Há dúvidas sobre quem passou a usar a palavra no contexto da computação, sendo que Alan Turing é um dos citados, e Paul Niquete reivindica ter usado o termo ao falar desde 1953. Normalmente o termo é associado a sua primeira publicação, por Tukey (1958). Um registro anterior, de 1956, em um artigo de Richard R. Carhart(Niquete, 2006) usa a palavra software com a acepção de pessoas envolvidas com o computador. Em francês existe a palavra *logiciel*, que os portugueses traduziram para logiciário. Este termo foi tentado no Brasil mas não chegou a ter força

Atualmente o termo software é associado não apenas ao programa de computador, mas também as suas estruturas de dados e sua documentação. Pressman e Maxim (2014) definem **software** como:

1. **instruções** (programas de computador) que quando executadas fornecem as funções, características e desempenho desejados;
2. **estruturas de dados** que habilitam o programa a manipular informação de forma adequada, e
3. **informação** que descreve a operação e uso do programa.

Já a IEEE define software como “programas de computador, procedimentos, e, possivelmente, documentação e dados pertinentes a operação de um sistema computacional”(IEEE, 2017b).

A Figura 2.1 mostra uma página do Sistema Acadêmico do CEDERJ e parte do código para sua criação. Apesar de HTML não ser uma linguagem de programação, é parte do que chamamos de software.

Um computador atual não funciona sem software. Na verdade, o computador que usamos para nossas tarefas diárias possui camadas e camadas de software para funcionar. É o software que instrui a máquina construída com circuito eletrônicos sobre os passos que ela deve seguir. Mesmo em uma CPU moderna, que normalmente vemos como *hardware*, já existe software e até sistemas operacionais instalados(Vaughan-Nichols, 2017).

## 2.1. O que é Software

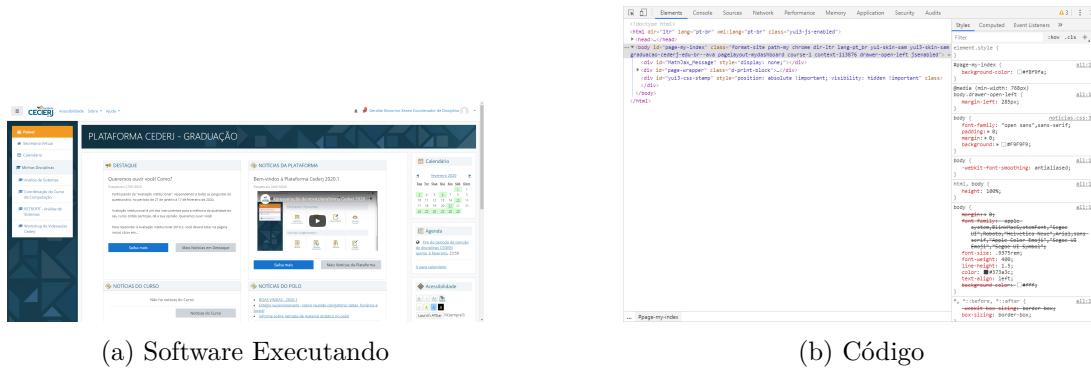


Figura 2.1.: Um software executando na web e parte do seu código.

Software é considerado um bem imaterial. É bastante difícil, por exemplo, calcular o seu **valor**. Quanto foi gasto para produzi-lo, quanto ele traz de lucro, quanto é gasto para mantê-lo, quanto seria gasto para substitui-lo seriam formar alternativas de fazer isso. Produtos como o sistema de reserva de passagens SABRE, originalmente criado para a American Airlines(Head, 2002), se tornaram tão importantes que se transformaram em negócios isolados(Levere, 2001), adquirindo valor próprio como negócio.

Outra característica do software é que seu custo de desenvolvimento é basicamente composto pelo esforço das pessoas envolvidas em produzi-lo. Software não exige ou consome matéria prima, e o uso de máquinas para produzi-lo é basicamente proporcional ao número de computadores necessários para as pessoas realizando tarefas, mas uns poucos computadores necessários para manter atividades automáticas. Isto não significa, porém, que seja barato desenvolver software, ao contrário, os custos de produção e manutenção de software são bastante altos e podem explodir se o projeto não for bem gerenciado. Esta é uma das preocupações principais da Engenharia de Software.

Software pode ser encontrado de várias formas: texto em arquivos digitais ou impressos, desenhos, arquivos binários ou em outros formatos, bits na memória do computador, documentações em diferentes mídias, como texto, e vídeos. Este documento, por exemplo, é possui uma versão digital no formato PDF que contém, dentro dele, programas em PostScript que instruem como deve ser impresso ou visualizado. Este é um exemplo da ubiquidade do software, isto é, ele está presente ao nosso redor e não percebemos mais essa presença.

### 2.1.1. Como cobrar? A pergunta mais comum nos cursos de ES.

Uma expectativa muito comum do iniciante na Engenharia de Software é que ela vai poder ser usada facilmente para responder a pergunta: quanto devo cobrar pelo software que produzo? Esse valor, porém, não é discutido na área. A questão importante para a Engenharia de Software é o **custo**, não o **preço**. O preço depende de outros fatores, como a oferta e a demanda do mercado. Preços de produtos e serviços são estudados por

## *2. Introdução a Engenharia de Software*

outras áreas do conhecimento, como a Economia e o Marketing. Mesmo assim, é possível apresentar algumas informações sobre preço.

É verdade que muitas vezes o software é cobrado em função do seu custo, ao qual é adicionado uma margem de lucro. Então, quem escolher essa forma de trabalho vai encontrar, nesse livro, algumas informações úteis.

O custo de software poder ser previsto ou calculado de várias maneiras, como por meio de horas trabalhadas, linhas de código, relatórios e telas, as estatísticas que permitem alguma comparação estão relacionadas a uma medida conhecida como Pontos de Função, que será tratada no Capítulo 15. No caso da previsão, sempre haverá uma incerteza, tanto por motivos da essência do software, quanto pela capacidade das técnicas. Além disso, todo projeto tem riscos. A Engenharia de Software tem entre suas metas aumentar a precisão das previsões e diminuir o risco de projeto.

Nesta forma de cobrança, é preciso prever o custo de alguma forma, e isto é feito em função do tamanho do esforço necessário para produzir o software, que pode ser equacionado a partir da funcionalidade entregue ao cliente. A medida mais aceita em todo mundo, principalmente pelos governos e órgãos regulatórios e de padronização é o Ponto de Função (IFPUG, 2010). Esse assunto será tratado no Capítulo 15. No momento é razoável saber que um relatório simples custa 4 pontos de função.

No mercado mundial, segundo Czarnacka-Chrobot (2012), com dados do ISBSG, um ponto de função custa entre US\$ 300 e US\$ 1000, com uma média de US\$750, mas foram encontrados projetos com valores desde US\$ 17 até US\$ 2727 na base consultada, incluindo os *outliers*. São dados que incluem custos não só de desenvolvimento, mas também de apoio.

Com esses mesmos dados, o preço por hora varia entre US\$60 até US\$105, com uma média de US\$ 80, havendo extremos entre US\$7 e US\$ 570.

Um ponto de função pode custar entre US\$ 1600 em média no Japão, até US\$ 125 em média na Índia. Não é de se espantar que tanto software seja terceirizado para a Índia. Em relação ao tamanho dos projetos, é comum que quanto maior o software, maior o preço do ponto de função.

No Brasil, devido a regulamentação do governo, software é muitas vezes precificado de duas formas: pelo uso de Pontos de Função (PF) e por Unidades de Serviço Técnico (USTs). USTs são acordadas por projeto, havendo uma validação com a prática do próprio governo em outros projetos. Os preços oscilam entre R\$ 250,00 até R\$ 1.500,00 (FATTO, 2020). É importante frisar que existe um regra de contagem de Pontos de Função destinada aos contratos do governo brasileiro que vai além do modelo original ou do padrão IFPUG (Brasil, 2018).

Isso significa que, para o governo brasileiro, uma tela com um relatório simples, como a quantidade de pedidos de aposentadoria por estado, por exemplo, pode custar algo entre R\$ 1.000,00 e R\$ 6.000,00.

A variação de preços é explicada por motivos como vários fatores de tecnologia, incluindo linguagem de programação, requisitos de operação continuada e de segurança, e ainda variações do mercado, como a oferta e demanda de profissionais, e outras.

## 2.2. Por que software é importante

Basicamente, software, na forma de dados e programas de computador, é o que hoje faz funcionar, monitora e controla todo o mundo. Se no início os computadores eram máquinas extremamente pesadas e caras que viviam em porões resfriados, trabalhando em modo *batch*, hoje praticamente todo equipamento que usamos possui alguma automação, seja das grandes usinas geradoras de energia a cafeteira elétrica que usa essa energia.

### Como tudo começou

O primeiro programa de computador, criado para um computador mecânico que nunca ficou pronto, foi feito por Ada Augusta King, nascida Byron, condessa de Lovelace. Ela criou e publicou um programa para a Máquina Analítica de Babbage, mostrando suas potencialidades.

Toda infraestrutura depende da geração de energia, que é planejada e controlada por meio de software, em vários níveis da produção. No Brasil, por exemplo, a ONS, o Operador Nacional do Sistema Elétrico, tem sistemas sofisticados que sabem o status de cada peça usada na transmissão de energia. Toda informação é trocada de forma digital, dos telefones às grandes redes de televisão, passando pela internet propriamente dita. Automóveis chegam a ter dezenas de processadores. Aparelhos domésticos podem se conectar à internet.

Por exemplo, a foto da Figura 2.2 mostra a sala de controle de uma usina de energia nuclear de tecnologia original americana, construída antes da massiva transformações das estruturas de comando e controle dessa indústria para software. É possível ver nessa foto dezenas de controles de hardware, porém também é possível ver que parte dos controles já foi modernizada para sistemas de software.

Assim, dentro dessa realidade, toda a economia é extremamente dependente do funcionamento de programas de computador de diferentes graus de sofisticação. Um freio ABS, por exemplo, possui software em várias camadas, uma servindo como defesa dos erros que as outras podem cometer.

Outra caracterização da importância pode ser dado pelo tamanho da indústria de software mundial, avaliada em 2014 em 407 bilhões de dólares pelo Gartner group (Group, 2014), enquanto toda a indústria de TI chegaria a 5.3 trilhões de dólares em 2020 (CompTIA, 2019). Além disso, em 19 de maio de 2021, cinco das seis maiores empresas em valor de mercado eram empresas de tecnologia da informação cujo principal diferencial é seu software: Apple, Microsoft, Amazon, Alphabet e Facebook. Essas são enormes

## 2. Introdução a Engenharia de Software



Figura 2.2.: Sala de controle de uma usina nuclear. Fonte: foto de Rodrigo Costa dos Santos

empresas de software de uso geral, existe um mercado ainda para software dedicado, como os que controlam um aparelho médico ou fazem uma usina de energia funcionar.

O custo de software geralmente excede o de hardware, até mesmo em sistemas complexos. Além disso, várias funções que antes eram feitas por hardware, hoje são feitas por software, o que apresenta vantagens, mas também aumenta a complexidade do que pode ser feito e consequentemente aumenta a complexidade do software, e também gera novos riscos para os sistemas.

Outra tendência é a virtualização do hardware. Isso significa que funções típicas de hardware são repassadas para camadas de software, sendo que isso acaba afetando também a forma de se pensar o hardware. A Figura 2.3 mostra uma arquitetura que pode ser encontrada em um servidor hoje em dia<sup>1</sup>.

### 2.2.1. A Ubiquidade do Software

Imagine uma pessoa qualquer em uma cidade moderna qualquer. Vamos chamá-la de João. João acorda porque toca um alarme em um relógio digital. Esse relógio está ligado na rede elétrica, que é controlada por software. O relógio contém software dentro dele. João levanta e liga a TV digital, que contém software, para ver as notícias. Todos os equipamentos da estação de notícias são digitais e contêm software. Ele olha seu celular, com sistemas operacionais e aplicativos altamente sofisticados, para verificar mensagens que vieram em uma rede de dados que contém camadas e camadas de software. Em 10 minutos de vida, João utilizou provavelmente centenas de milhões de linhas de código sem ao menos perceber sua dependência da indústria de Tecnologia da Informação.

---

<sup>1</sup>É interessante notar que também existe um movimento contrário, de trazer funções desenvolvidas em software para o hardware, permitindo por sua vez que o software possa ser mais sofisticado, em um ciclo de *feedback* positivo, o que aconteceu por exemplo na sofisticação das placas gráficas(Savage, 2020)

## 2.2. Por que software é importante

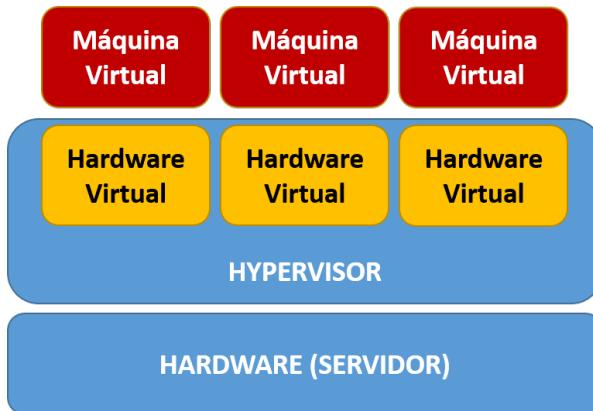


Figura 2.3.: Esquema da virtualização do hardware.

A verdade é que o software, e toda a Tecnologia da Informação, se tornou algo ubíquo. Essa palavra indica um tipo de ideia ou objeto que é pervasiva, está em todo lugar, ao ponto que não percebemos mais sua presença.

## 2.3. Quais os riscos que software traz?

Sendo tão importante, é claro que software também traz riscos às organizações e às pessoas. As duas maiores facetas desse risco são as perdas de vida humana, animais ou bens, por erro do software, e as grandes perdas financeiras causadas por software ou por projetos de software.

Acidentes de aviação onde houve influência do software tiveram grande repercussão na mídia, como o caso dos aviões Boeing 737 Max. Um dos casos mais discutidos tecnicamente, ligado a perda de vidas humanas é o do equipamento de terapia por radiação Therac-25 (Levenson e Turner, 1993), onde uma falha em um sistema de proteção feito em software causou que doses altíssimas fossem aplicadas a várias pessoas, causando morte ou lesões em pelo menos 6 pessoas.

Alguns casos são interessantes de citar. O primeiro lançamento do foguete europeu Ariane V, em 1996, resultou em uma autodestruição automática devido a uma exceção não tratada no software (Lions, 1996), não só atrasando o projeto em anos, mas também criando um problema para a imagem do projeto e impactando o prêmio do seguro de lançamento de satélites para toda a indústria. No Brasil, a Volkswagen, em 2006, anunciou o *recall* de 123 mil veículos para trocar um software que controlava o funcionamento dos componentes eletrônicos dos automóveis (Folha Online, 2006). No caso, alguns dos automóveis tinham o chip dentro do motor, o que obrigava a abri-lo, operação complicada e que ninguém gosta que seja feita em seu carro.

Já em prejuízos financeiros os casos se repetem ano após ano. Pequenos problemas de software, como aplicativos de celular parando de funcionar, ou erros de cálculo em fórmulas que usamos em planilhas eletrônicas, podem ter um grande valor agregado na economia, se somadas todas as horas de trabalho perdidas ou ainda os prejuízos reais causados. A Tricentis (2019) estimou que as perdas devidas a falhas de software somaram US\$1,72 trilhões em 2017, afetando 3,68 bilhões de pessoas e com uma perda acumulada de 268 anos. Já Krasner (2021) identificou um custo total de US\$ 2.08 causado por baixa qualidade de software, com mais US\$ 1.31 trilhões a serem gastos no futuro para correções.

Projetos inacabados com altos custos desperdiçados são também bastante comuns. Um dos maiores fracassos relatados aconteceu nas décadas de 80 e 90, quando a Administração Federal de Aviação americana (FAA) tentou criar um novo sistema de controle do tráfego aéreo. Reconhecido como ambicioso demais, um dos seus requisitos era de 99.99999% de confiabilidade, o sistema foi planejado para custar, na época, US\$ 500 por linha de código, o que já era cinco vezes mais do que o padrão da indústria para processos bem gerenciados. Em 1995 o custo já tinha chegado de US\$ 700 a US\$ 900 por linha de código. Inicialmente orçado em US\$ 2.5 bilhões, em 1994 as estimativas subiram para US\$ 7 bilhões e o projeto foi abandonado, sendo que apenas algumas de suas partes foram adotadas como partes de projetos menores (Gibbs, 1994).

## 2.4. Por que é difícil fazer software?

Entre outros grandes fracassos podemos adicionar o *FBI* tendo que jogar fora US\$ 170 milhões em um sistema de apoio a identificação de terroristas, a *Ford Motor Company* abandonando um projeto de US\$ 200 milhões de um novo sistema integrado de compras e o *McDonald's* declarando US\$ 170 milhões em prejuízo por um software para automatizar toda sua operação (Carr, 2005).

Não apenas erros, mas falhas de design na interface, e também falta de dados em software funcionando, também já influenciaram acidentes graves. Os acidentes com aviões B757, do vôo AA965 em Cali, de 20/12/1995, com 159 fatalidades, e do vôo ALW301 em *Puerto Plata*, de 6/2/1996, com 189 fatalidades, foram, em parte, associados a problema na interface homem-computador (Ladkin, 1996).

Problemas de software podem também fazer com que equipamentos não cumpram sua missão de proteção. O acidente do vôo 801 da *Korean Air*, em 3/12/1984, em Guam, que causou 228 mortes, por exemplo, poderia ter sido prevenido se não houvesse um erro no sistema do Radar de Altitude Segura Mínima (Bellovin, 1997).

Existem muitos outros riscos que são relatados diariamente. Equipamentos podem ser controlados por pessoas estranhas a sua operação, câmeras podem ser monitoradas por pessoas sem esse direito, dados pessoais podem ser obtidos de forma ilegal ou legal mas sem o conhecimento dessas pessoas, com grandes ameaças a privacidade, defeitos ou *bugs* podem fazer equipamentos pararem de funcionar a qualquer momento, etc. Uma boa fonte de notícias sobre o assunto é o *The Risks Digest: Forum on Risks to the Public in Computers and Related Systems*, moderado por Peter G. Neumann<sup>2</sup>.

## 2.4. Por que é difícil fazer software?

A Engenharia de software nasceu porque fazer software é difícil. O nome foi estabelecido em uma conferência da OTAN em 1968 cujo objeto era descobrir como resolver um problema de engenharia: o projeto, a produção e a manutenção com sucesso de sistemas de software úteis (Naur e Randell, 1969).

Dijkstra descreveu bem o problema de desenvolver software em 1972, na sua palestra de aceitação do prêmio Turing. Se colocando no lugar de um programador do futuro, ou seja, aproximadamente agora, ele conta: “quando não havia máquinas, programar não era um problema de forma alguma; quando nós tínhamos poucos computadores fracos, programar se tornou um problema, e agora nós temos computadores gigantescos, programar se tornou um problema gigantesco” (Dijkstra, 1972). Ou seja, como temos computadores superpoderosos podemos tentar resolver problemas difíceis.

Além de difíceis, grande parte dos problemas que tratamos são conhecidos como **wicked problems**, cuja tradução poderia ser problemas perversos. Esses problemas se caracterizam por (Conklin, 2005):

---

<sup>2</sup><http://catless.ncl.ac.uk/Risks/>

## 2. Introdução a Engenharia de Software

- Não serem entendidos até que uma solução seja desenvolvida
- Não haver uma regra de parada para a solução
- Suas soluções não serem certas ou erradas, mas sim melhores, aceitáveis ou não boas o suficiente
- Serem essencialmente únicos e novos
- Cada solução possível ser uma operação única e com consequências
- Não possuírem soluções alternativa

Somam-se a isso outros desafios.

Um deles é a necessidade de atender as demandas das pessoas em um contexto social, compondo diferentes partes interessadas. Isso implica em problemas de comunicação, política organizacional, etc. A complexidade social (Conklin, 2005) geralmente ocorre junto com um problema perverso (*wicked problem*) e é função do número e da diversidade de partes interessadas envolvidas em um projeto.

### Partes Interessadas

Esse tópico será tratado detalhadamente no Capítulo 5. Um definição simples é “toda pessoa, organização ou entidade que pode ser, ou acredita que pode ser, beneficiada ou prejudicada por um projeto”.

Outro problema é o técnico. Projetos de software envolvem pessoas trabalhando em um grande número de partes. Um software de funcionalidade relativamente simples pode ter milhares de linhas de código. Projetos podem durar anos. Tudo isso implica na necessidade de técnicas de gestão de projeto, garantia de qualidade, e outros assuntos que compõe a Engenharia de Software.

Em especial, não só a complexidade de um projeto de software aumenta de forma mais que linear com a quantidade de funcionalidade entregue, quanto os erros de estimativa aumentam mais ainda (Gibbs, 1994). Isso é bem diferente de grande parte da indústria, onde produzir mais pode inclusive baixar os preços.

Para entender o tamanho de um software atual, é possível lembrar que foram necessárias 145.000 linhas de código para o software de orientação da Apollo XI, já o sistema Android versão 2.2 continha 12 milhões de linhas de código (P. Johnson, 2012). A Figura 2.4 mostra o tamanho comparativo de alguns software conhecidos, que chega a 86 milhões de linhas de código para o Mac OS X 10.4. Isso mostra que mesmo um programa pequeno que funcione em sistema operacional como Android, Windows ou MacOS depende de milhões de linhas de código funcionarem corretamente. Para comparação, uma das máquinas mais complexas existente no mundo, o Boeing 747, tem aproximadamente 6 milhões de partes distintas, e um automóvel apenas 30 mil.

Outra questão importante do software é que dificilmente ele se mantém constante ao longo do tempo quando em operação. Dois são os motivos dessa mudança constante: a descoberta de *bugs* que precisam ser consertados e a alteração dos requisitos originais com a evolução do ambiente em que funcionam. Por exemplo, em um software de gerência de

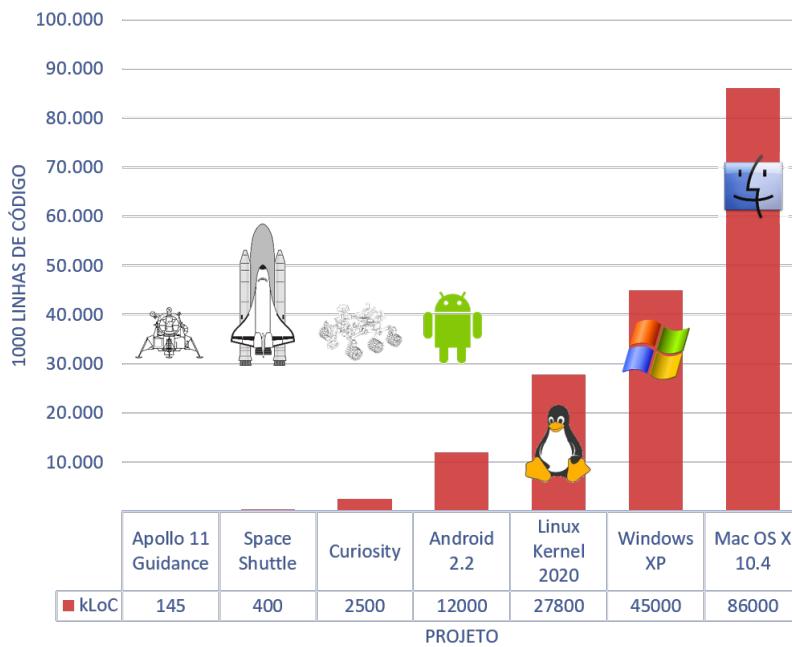


Figura 2.4.: Tamanho de código de alguns projetos conhecidos.

recursos humanos, qualquer alteração nas leis trabalhistas pode exigir uma modificação no software para que fique de acordo com as novas leis. O problema é que qualquer modificação no software tem uma chance de inserir novos *bugs*.

A maioria dos autores de Engenharia de Software concorda que a curva de falha do software, que temos dificuldade de tratar, Figura 2.6, é bem diferente da curva de falha do hardware, Figura 2.5, a qual já é bem modelada estatisticamente.

### 2.4.1. O Relatório CHAOS

Pesquisas de campo também mostram que muitos projetos de software falham ou acabam com problemas. Desde de 1994 o *Standship Group* realiza uma pesquisa chamada “CHAOS Report” que mostra o estado dos projetos de software. As medidas básicas para determinar a resolução de um projeto são (The Standish Group, 2015):

- dentro do **orçamento**;
- dentro do **prazo**, e
- satisfatório em relação a **funcionalidade**.

Essa pesquisa divide os projetos. por resolução, em: **fracassos**, que foram cancelados ou não foram usados, **com problemas**, em uma ou duas das três medidas básicas, e **bem sucedidos**<sup>3</sup>. Na versão de 2015, com uma base de 2011 a 2015, os percentuais

<sup>3</sup>Os termos originais são *failed*, *challenged* e *successful*.

## 2. Introdução a Engenharia de Software

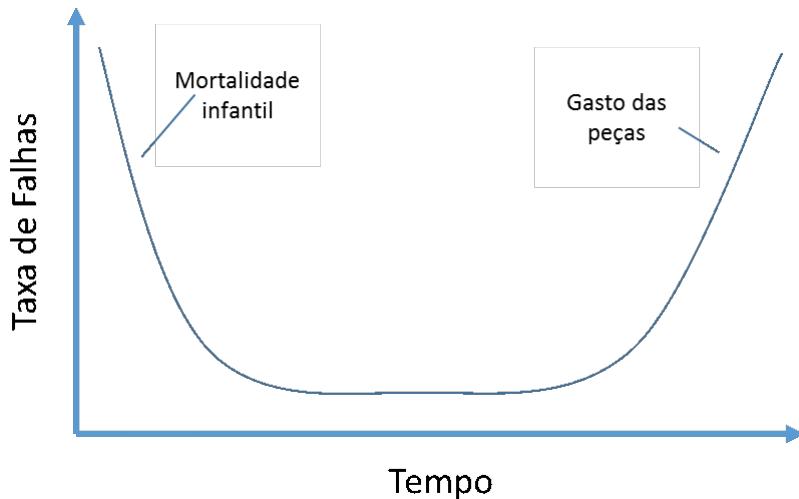


Figura 2.5.: Curva de falha do hardware. Fonte: (Pressman e Maxim, 2014)

mostraram números preocupantes, que podem ser vistos na Tabela 2.1, principalmente para projetos grandes.

Tabela 2.1.: Resolução de projetos de software por tamanho e por método de desenvolvimento, para mais de 10.000 projetos. Fonte:(The Standish Group, 2015)

Tamanho	Método	Bem Sucedido	Com Problemas	Fracassos
Grande	Ágil	18%	59%	23%
	Cascata	3%	55%	42%
Médio	Ágil	27%	62%	11%
	Cascata	7%	68%	25%
Pequeno	Ágil	58%	38%	4%
	Cascata	44%	45%	11%

Como podemos ver, ainda na Tabela 2.1, a taxa de fracassos aumenta muito com o tamanho, principalmente para projetos feitos com a metodologia mais tradicional, em Cascata. Além disso, a taxa de sucesso para projetos médios e grandes está abaixo de 30% mesmo para projetos feitos com a metodologia ágil, e menor que 10% para a metodologia tradicional. Isso é uma constatação da dificuldade de fazer projetos de software.

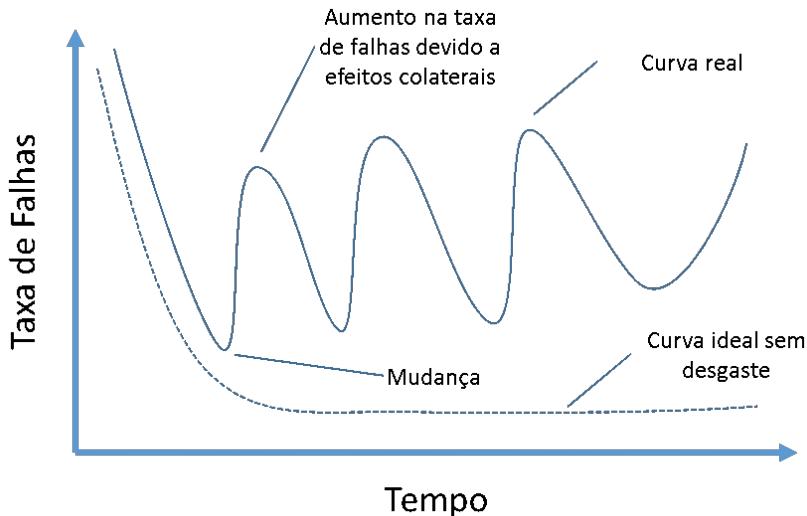


Figura 2.6.: Curva de falha do software. Fonte: (Pressman e Maxim, 2014)

## 2.5. O que é Engenharia de Software

**Engenharia de Software** é “a aplicação de uma abordagem sistemática, disciplinada e quantificável ao desenvolvimento, operação e manutenção de software, i.e., a aplicação da engenharia ao software” (IEEE Computer Society, 2014a).

O termo foi criado em torno de 1968, perante a necessidade de melhorar o desenvolvimento de software, que desde o início se mostrou algo bastante difícil. Pesquisas mostram não só que muitos projetos de software falham, parcial ou totalmente, quanto o aumento drástico da probabilidade de fracasso de um projeto software com o aumento da complexidade do mesmo (The Standish Group, 1994).

Segundo o SWEBOK, a “Engenharia de Software é a profissão preocupada com a criação e manutenção de aplicações de software pela aplicação de tecnologias e práticas da Ciência da Computação, Gerência de Projetos, Engenharia, domínios de aplicação e outros campos”(IEEE Computer Society, 2014a).

Uma abordagem mais pragmática é pensar a Engenharia de Software como o estudo das formas e as próprias formas de desenvolver software de maneira custo-efetiva. Fornecer os maiores benefícios possíveis dentro de um conjunto de restrições é um dos principais objetivos da Engenharia que se aplica também a Engenharia de Software.

As metas da Engenharia de Software incluem atender o cliente que precisa do software, gerar valor para as organizações que produzem e utilizam o software e deixar satisfeitos, também, aqueles que produzem o software.

## 2. Introdução a Engenharia de Software



Figura 2.7.: Pirâmide que explica a construção de conceitos na Engenharia de Software.

### 2.5.1. Uma Visão da Engenharia de Software

Pressman e Maxim (2014) construíram uma pirâmide de 5 níveis para explicar a Engenharia de Software, em uma abordagem bastante técnica. Porém, é necessário mostrar também o que motiva essas técnicas, o que é parte significativa do problema da Engenharia de Software.

Desse modo, a Engenharia de Software pode ser vista por uma pirâmide de 7 níveis, Figura 2.7, onde tudo se baseia em atender as **partes interessadas**, pela agregação de **valor**. Isso só pode ser feito por meio de produtos e serviços de **qualidade**, o que exige **processos**, onde são usados **métodos**, de acordo com **práticas**, que exigem **ferramentas**. Durante o livro, todos esses assuntos serão tratados.

## 2.6. Por que a Engenharia de Software é necessária?

Precisamos da Engenharia de Software porque não é possível fazer software “de qualquer jeito”. O Ciclo de Vida de um software, de qualquer tamanho, é um projeto que se inicia na descoberta do que se deve fazer e termina quando o software, depois de entregue e usado, é retirado do funcionamento. A fase de suporte e manutenção normalmente dura muito mais que a de desenvolvimento. Para tudo isso é preciso usar técnicas típicas da engenharia, como a construção de modelos abstratos do software que deve ser criado, estimativas de prazo e custo, garantia da qualidade do produto, etc.

### 2.6.1. Diferenças para outras engenharias

Mas o que há de diferente entre Engenharia de Software e outras engenharias? A princípio, fazer software poderia ser considerado até mais fácil, já que na maioria das engenharias, após construirmos um artefato, temos que reproduzi-lo por um processo industrial. Software pode ser simplesmente copiado, com uma taxa baixíssima de falha

## *2.7. Qual o Segredo do Sucesso de um Projeto de Software*

na cópia e, mesmo em mídias que degradam, de fácil verificação. Assim, a Engenharia de Software se parece mais com a parte das engenharias que costumamos chamar de desenvolvimento do projeto ou desenvolvimento do protótipo final.

Por outro lado, em um projeto de software cada linha programada tem importância. E a quantidade de linhas programadas, ou itens que devem ser analisados separadamente, e ainda a interação entre elas, seja por meio de código ou dados, é enorme. Assim, construir software é, de várias maneiras, muito mais complexo que construir uma casa. Se um tijolo da casa não tem um pequeno pedaço, não há problema nenhum. Se um grão de pó de um elemento químico em um processo químico tem um peso inferior a média, o problema certamente também não existirá. Mas troque uma letra <sup>4</sup> em um programa de computador e você pode ter um programa que nem compila, ou funciona, porém, executa de forma absolutamente errada<sup>5</sup>.

Outro problema é a dificuldade de saber o que é realmente necessário, devido aos problemas naturais de comunicação entre as pessoas, principalmente usuários e desenvolvedores. Além disso, muitas vezes não se sabe o que deve ser feito, mas sim o problema que deve ser resolvido, o que implica em processos de investigação ainda mais complicados.

## **2.7. Qual o Segredo do Sucesso de um Projeto de Software**

Como vimos, desenvolver software é difícil. A área de Engenharia de Software, então, se preocupa em encontrar formas de melhorar esse desempenho. Segundo o mesmo The Standish Group (2015), se forem utilizadas Metodologias Ágeis, o índice de fracasso cai de 42% para 23% em projetos grandes, e o de sucesso sobe de 3% para 18%.

Esse mesmo relatório relata as razões de sucesso de projetos de software, apresentadas na Tabela 2.2. Observando a Tabela, é possível perceber que os principais fatores não estão ligados diretamente a Processos, Métodos e Ferramentas, porém eles precisam ser alcançados com o uso das técnicas de Engenharia de Software, principalmente o Envolvimento do Usuário e a Otimização, que começa com a gerência de escopo com base no valor de negócio (Hastie e Wojewoda, 2015)

Ainda analisando a Tabela 2.2 e usando conhecimentos de Engenharia de Software, podemos dizer que o envolvimento do usuário é melhorado com o uso de processos ágeis.

---

<sup>4</sup>A quantidade mínima e discreta de um programa escrito em uma linguagem de programação de terceira geração ou maior

<sup>5</sup>Em um exemplo que aconteceu com o autor, dois espaços de uma identação em Python fizeram que um programa gerasse os dados errados durante dias, até que o erro fosse descoberto.

## 2. Introdução a Engenharia de Software

Tabela 2.2.: Fatores de Sucesso de um Projeto de Software. Fonte:(The Standish Group, 2015)

Fator de Sucesso	Peso Relativo
Patrocínio Executivo	15
Maturidade Emocional	15
Envolvimento do Usuário	15
Otimização	15
Recursos Capacitados	10
Arquitetura Padrão	8
Processo Ágil	7
Execução Modesta	6
Experiência em Gerência de Projetos	5
Objetivos Claros de Negócio	4

## 2.8. O Corpo de Conhecimento da ES

Uma das publicações mais importantes da Engenharia de Software é o *Guide to the Software Engineering Body of Knowledge (SWEBO(R))*. Esse guia, construído colaborativamente por dezenas de pesquisadores ao redor do mundo e mantido pela IEEE Computer Society, define os conhecimentos relativos a área, dividindo-os em 15 áreas do conhecimento(IEEE Computer Society, 2014a), como descritas na Figura 2.8.

Cada área do conhecimento é, por sua vez, subdividida em tópicos e subtópicos. O SWEBOK serve como definição do escopo do que é geralmente reconhecido como Engenharia de Software e ainda como referência primária para cada tópico, devido as citações que ele contém, além de possuir uma referência cruzada entre seus tópicos e as normas ISO/IEC a eles relacionadas.

Analisando criticamente, o SWEBOK é um registro, feito em um momento específico, do que se acordou, entre uma grande quantidade de pesquisadores, do que seriam os tópicos relacionados a Engenharia de Software. Com a evolução da própria Engenharia de Software, o projeto vai ficando desatualizado.

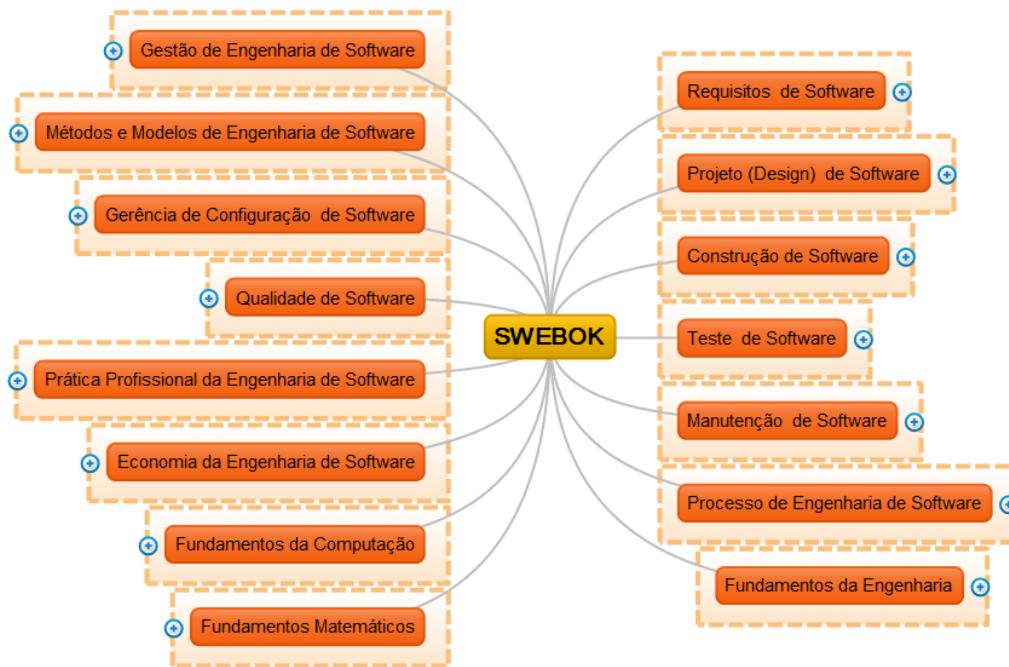


Figura 2.8.: Mind map para as áreas de conhecimento da Engenharia de Software:  
Fonte:(IEEE Computer Society, 2014a)

## 2.9. As normas da Engenharia de Software

Uma parte do conhecimento de Engenharia de Software foi normatizada por instituições nacionais e internacionais, principalmente a *International Standards Organizations* (ISO) e o *Institute of Electrical and Electronics Engineers* (IEEE)<sup>6</sup>. No Brasil, as normas são criadas, normalmente por tradução, pela Associação Brasileira de Normas Técnicas (ABNT) e são numeradas com o código NBR.

As normas geralmente se referem a conceitos estabelecidos, isto é, raramente há normas sobre inovações na Engenharia de Software, porém, nos últimos anos o processo de criação tem se tornado mais dinâmico.

De grande influência na indústria em geral são as normas de qualidade (família ISO 9000), que foram especializadas para software primeiro na família ISO 9216 e depois na família ISO 25000.

<sup>6</sup>Pronunciado i-três-é ou *ai-Triple-i*

## 2. Introdução a Engenharia de Software

### 2.10. SEMAT

Um movimento interessante na Engenharia de Software é o SEMAT – *Software Engineering Method and Theory*. Fundado por Ivar Jacobson, Bertrand Meyer e Richard Soley em 2009, o SEMAT se propõe a “refundar a Engenharia de Software como uma disciplina rigorosa baseada em uma teoria geral da Engenharia de Software e um arcabouço de processo unificado”(Jacobson, Lawson e Ng, 2019; Jacobson, Ng et al., 2013).

Seu principais resultados são o Kernel, composto de Alphas, *Abstract Level Progress Health Attribute*, representações das coisas essenciais que a Engenharia de Software trabalha, Espaços de Atividade e Competências. As Figuras 2.9, 2.10 e 2.11 apresentam, respectivamente, uma representação dessa contribuição.

O SEMAT também define métodos e práticas. Um **método** “provê um guia para todas as coisas que você precisa fazer quanto está desenvolvendo e mantendo um software”(Jacobson, Lawson e Ng, 2019). Métodos são compostos por **práticas**, que são essas coisas que são feitas, ou mini-métodos. Práticas também podem ser compostas por outras práticas(Jacobson, Lawson e Ng, 2019). Segundo Jacobson, Lawson e Ng (2019), enquanto existem muitos métodos no mundo, o conjunto de prática é bem menor. Elas também são reusáveis, podendo aparecer em vários métodos.

#### 2.10.1. O Kernel Essencial

O Kernel Essencial é estruturado em 3 áreas: Cliente, Solução e Empreendimento. Dentro dessas áreas estão os Alphas, como mostra a figura a 2.9.] Os Alphas são as principais dimensões de um processo de software(Jacobson, McMahon e Racko, 2015).

As Oportunidades do Kernel Essencial podem ser diretamente associadas ao conceito de Valor, já que são oportunidades de agregar valor as Partes Interessadas.

As **partes interessadas**, estudadas no Capítulo 5, são “pessoas, grupos ou organização que afetam ou são afetadas pelo sistema de software”(SEMAT Inc., 2019). São as partes interessadas que fornecem **oportunidades**, o “conjunto de circunstâncias que tornam apropriada o desenvolvimento ou mudança de um sistema de software”.

Essas **oportunidades** focalizam os **requisitos**, que são “o que o software precisa fazer para endereçar as oportunidades e satisfazer as partes interessadas”(SEMAT Inc., 2019). O **sistema de software** deve atender os requisitos, sendo o “produto primário do empreendimento de engenharia de software”(SEMAT Inc., 2019). O **time** é o “grupo de pessoas ativamente engajada no desenvolvimento, manutenção, entrega ou suporte de um sistema de software”(SEMAT Inc., 2019). É o time que executa e planeja o **trabalho**, pela aplicação de um ou mais **modos de trabalho**.

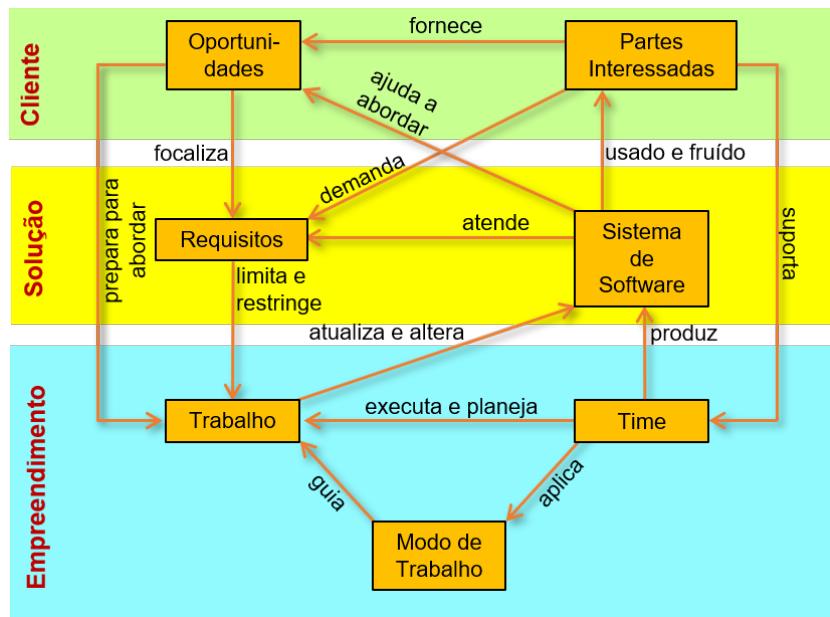


Figura 2.9.: O Kernel Essencial do SEMAT. Fonte:(Jacobson, Ng et al., 2013)

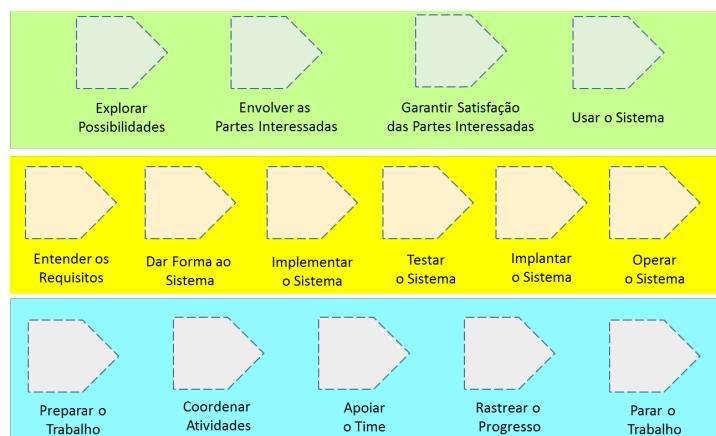


Figura 2.10.: Os Espaços de Atividades no Kernel do SEMAT. Fonte:(Jacobson, Ng et al., 2013)

## 2. Introdução a Engenharia de Software



Figura 2.11.: As Competências no Kernel do SEMAT. Fonte:(Jacobson, Ng et al., 2013)

## 2.11. Quem é o Engenheiro de Software

Não é fácil, hoje em dia, definir exatamente o que faz um Engenheiro de Software, porque a área é muito ampla, como se pode ver na Figura 2.8.

Uma carreira profissional na Engenharia de Software pode começar com funções ligadas a testes e controle de qualidade, ou a programação. A evolução técnica e pessoal do profissional vai levá-lo a cargos de maior responsabilidade técnica e gerencial, como cuidar da arquitetura de software ou gerenciar equipes de desenvolvimento. Com o tempo, deve subir mais na hierarquia e ser responsável por projetos ou mesmo portfólios de projetos.

A IEEE Computer Society lançou em 2014 um documento que fala sobre as competências esperadas de um Engenheiro de Software( IEEE Computer Society, 2014b). Ele divide as competências em cinco grandes áreas, que são detalhadas no documento:

1. Habilidades Técnicas
2. Habilidades e Atributos Comportamentais
3. Habilidades Cognitivas
4. Conhecimento Necessário
5. Disciplinas Relacionadas

Cada uma dessas habilidades possui atividades, que podem ser classificadas em 5 níveis de competência IEEE Computer Society (2014b):

1. **Técnico**, que segue instruções;
2. **Profissional (*Practitioner*) de Nível Inicial**, que faz atividades com supervisão;
3. **Profissional (*Practitioner*)**, que faz atividades com pouca ou nenhuma supervisão
4. **Líder Técnico**, que lidera equipes e indivíduos no desempenho das atividades, e
5. **Engenheiro de Software Sênior**, que serve de Engenheiro Chefe.

Além disso, se espera que todo Engenheiro de Software seja um instrutor e mentor de outros( IEEE Computer Society, 2014b).

## 2.12. Exercícios

### Exercício 2.1:

Procure outra definição para Engenharia de Software. Discuta as diferenças entre essa definição e as dadas no texto.

### Exercício 2.2:

Procure acidentes ou problemas causados por software funcionando incorretamente? Procure problemas na indústria espacial e médica, por exemplo.

## *2. Introdução a Engenharia de Software*

### **Exercício 2.3:**

Procure prejuízos financeiros causados por projetos de software que sofreram algum problema? Procure relatos sobre grandes projetos governamentais ou de empresas que foram interrompidos ou não tiveram sua funcionalidade completa devido a problemas no software.

### **Exercício 2.4:**

Investigue prejuízos causados pelo mal uso de um software funcionando corretamente, por exemplo, quando planilhas Excel foram mal feitas. Ou acidentes de avião causados pelo erro de digitação. Explique como o software, mesmo funcionando corretamente, pode ter ajudado a causar o erro. Indique uma melhoria que poderia ter ajudado a alertar sobre o erro.

### **Exercício 2.5:**

Procure um caso onde um símbolo ou espaço errado fez com que um programa não funcionasse de forma drástica.

### **Exercício 2.6:**

Escolha um objeto ao seu redor que possua software e investigue o que esse software faz e onde ele está. Investigue o tipo de CPU usada por esse objeto.

### **Exercício 2.7:**

Software também pode ser gerado para produzir erros em máquinas e equipamentos, passando a ser chamados de vírus ou cavalos de Tróia. Procure o relato de um vírus gerado intencionalmente para causar prejuízos específicos para uma organização, ou algo mais genérico que tenha provocado prejuízos a muitos (e talvez lucros a seus criadores).

### **Exercício 2.8:**

Procure algumas descrições de empregos para Engenheiros de Software e faça um relato de suas atribuições. Avalie o grau de consistência dessas atribuições entre os empregos. Avalie também o que é mais técnico e o que é mais gerencial.

### **Exercício 2.9:**

Procure o documento ( IEEE Computer Society, 2014b) e entenda as habilidades desejadas de um Engenheiro de Software.

# 3

## Valor

### Conteúdo

---

3.1.	Conceituação Genérica de Valor . . . . .	32
3.2.	Valor na Economia . . . . .	33
3.3.	Valor em Marketing . . . . .	35
3.4.	Priorização Baseada em Valor para o Cliente . . . . .	41
3.5.	Valor Financeiro de Projetos . . . . .	48
3.6.	Valor em Software . . . . .	53
3.7.	Conclusão . . . . .	55
3.8.	Exercícios . . . . .	55

O objetivo de qualquer software é **entregar valor às partes interessadas**. O conceito de valor, porém, tem várias acepções. Mesmo que muitos métodos e autores se baseiem neste objetivo, raramente valor é definido. Assim, fornecedores e clientes se perguntam: o que é valor? Como posso medir valor?

A resposta pode depender da escolha de uma visão comum do significado de valor que depende de todas as partes interessadas do sistema, logo, é influenciada também por todo o contexto onde o projeto está inserido, incluindo o *background* das partes interessadas.

Este capítulo busca mostrar as várias facetas da ideia de valor. Começa com uma abordagem genérica e depois mostra alguns entendimentos da palavra valor na economia, no marketing e na área de software. Obviamente há alguma similaridade entre as definições, mas os diferentes interesses entre as áreas também fornecem um conhecimento mais amplo do que é valor e de como as pessoas entendem o que é valor.

### 3.1. Conceituação Genérica de Valor

A palavra **valor** pode ser usada em vários sentidos (Cairncross, 1951, pg. 140):

- o **valor moral**, por exemplo, consideramos um *valor* a liberdade, a honestidade;
- o **valor estético**, por exemplo, damos *valor* a obra de Camões;
- o **valor em uso**, ou seja, ligado a utilidade, a como uma coisa atende as necessidades de alguém, como o valor da água, que é alto para a existência de vida, mesmo ela sendo barata;
- o **valor de troca**, por exemplo, o preço de uma casa a ser vendida, e
- o **valor ideal de troca**, o preço que um comprador imaginário pagaria pela casa.

Procuramos entender a acepção de valor com relação aos assuntos ligados aos negócios, em busca de entender **o que é valor para uma organização que está adquirindo ou desenvolvendo um software**. Isto não significa, porém, que a presença de atributos éticos e estéticos em um software não agregue valor, justamente o contrário. É possível, inclusive, que uma organização procure um produto que faça as mesmas funções que o produto que ela já possui, mas que seja mais agradável de usar, valor estético, ou garanta algumas propriedades ligadas ao valor ético, como privacidade.

Em todo caso, mesmo que a organização busque originalmente outro tipo de valor, no momento do projeto é mais provável que a interpretação dada ao conceito seja, na prática, ligada a fatores econômicos, financeiros ou legais, de acordo com a situação.

Da forma mais geral, é possível entender valor como **uma qualidade atribuída a algo ou alguém**, mas essa definição é muito ampla. Beleza, por exemplo, também é uma qualidade atribuída. Neste quadro em especial, valor é uma qualidade relacionada com a **utilidade que um produto ou serviço tem para os que o consomem ou usufruem**.

A princípio, valor econômico é “a importância que um indivíduo dá a determinado bem ou serviço, seja para uso pessoal ou organizacional, seja para a troca” (Holanda Ferreira, 1986).

Essa definição do Dicionário Aurélio é interessante para software , porque já diz que o valor é ligado a importância dada ao bem ou serviço que está sendo recebido para o uso, ou sendo produzido para a troca.

Nas próximas seções veremos definições de valor nas seguintes áreas:

- Economia;
- Marketing;
- Projetos, e
- Software.

## 3.2. Valor na Economia

As tentativas de definir algo que pode ser chamado de valor na Economia tem, historicamente, também relação com o entendimento do preço dos objetos e serviços.

Na Economia Clássica, Smith, em 1776, primeiro reconhece que valor pode ter dois significados diferentes, um ligado a utilidade de um objeto particular, e outro ao poder de comprar outro bens. O primeiro seria o “valor de uso”, o segundo o “valor de troca”(Smith, 2003, Livro I, Cap. 4, para. 13)

Smith ainda discutiu valor, no sentido de preço, ou o preço relativo entre bens e serviços (Cairncross, 1951), com três abordagens (King e McLure, 2014), sendo que a terceira antecipou a teoria subjetiva de valor que apareceria mais tarde:

- o trabalho incorporado, que seria adequado as sociedades primitivas;
- a soma dos custos de produção, incluindo terra, capital e trabalho, mais adequado ao capitalismo, e
- a quantidade trabalho e o incômodo de adquirir, ou que é poupadão e pode ser imposto a outro, que é seu valor de troca (Strathern, 2003).

Podemos dizer que muitas empresas usam abordagens como essa para calcular o preço de seu software: calcular todos os custos de produção e somar um lucro, ou **markup**, sobre esse preço.

Também as organizações ou pessoas que estão pagando por um produto de software podem ser associadas a definição mais evoluída de Adam Smith: elas pagam para evitar algum incômodo.

Já em 1821, Ricardo (1996) propôs que para se saber o valor é necessário saber a utilidade, e que existiria um preço primário e natural, dado pela quantidade comparativa de trabalho necessário para a produção (King e McLure, 2014).

Novamente, a interpretação clássica de valor tem relação com a prática de desenvolvimento de software: utilidade para o usuário, custo de produção para o desenvolvedor.

### 3.2.1. Utilidade

**Utilidade** é o grau satisfação que obtemos do consumo de bens e serviços (Krugman e Wells, 2013). Não é possível medir utilidade de forma prática, ou seja, dizer que algo tem 100 ou 2354 de utilidade para alguém. Porém, teoricamente, se discute a **função utilidade**, que é individual. Isso significa que cada pessoa tem uma satisfação diferente consumindo um produto, e não é possível comparar utilidade entre pessoas.

De acordo com o **princípio da diminuição da utilidade marginal**, cada unidade adicional de bens ou serviços adiciona menos a utilidade total do que a unidade prévia. Esse é um princípio genérico. A verdade é que existem produtos que tem utilidade marginal que aumenta, por exemplo, se você precisa de uma quantidade  $X$  para atingir

### 3. Valor

um objetivo mínimo, a utilidade marginal não vai diminuir até você atingir  $X$  unidades, e pode até aumentar se for uma solução pior ter  $X - 1$  unidades (Krugman e Wells, 2013). Um exemplo seria construir um muro de tamanho pré-determinado com tijolos, a utilidade dos tijolos não vai diminuir até atingir a quantidade necessária para o muro. Porém se você quer um muro que garanta privacidade, sem especificar a altura, a utilidade dos tijolos vai começar a diminuir quando certo tamanho é atingido, referente a uma privacidade necessária, até que a privacidade máxima seja alcançada e a utilidade marginal seja zero.

Software é um desses casos. Para um software funcionar, é necessária uma funcionalidade mínima que permita que ele comece a funcionar, porém após essa funcionalidade mínima, cada funcionalidade a mais vai seguir esse princípio, até o ponto que o cliente não mais vai querer pagar por uma funcionalidade, porque não vai valer a pena, já que o investimento vai ser maior que o retorno.

Utilidade é uma coisa importante no valor de produtos de software. Basicamente, em todas metodologias que usam o conceito de valor, o que se pode deduzir das explicações dadas é que **o mais útil em um certo contexto tem mais valor e deve ser priorizado**. Em software, utilidade pode ser um sinônimo de valor em alguns projetos.

#### 3.2.2. Microeconomia Moderna e o Custo de Oportunidade

O **custo de oportunidade** indica “do que alguém deve desistir para obter o que deseja”(Greenlaw, Shapiro e Taylor, 2017), ou seja, indica a oportunidade perdida de se consumir ou usufruir outra coisa quando se consome ou usufrui de algo. É o “custo da próxima melhor alternativa” ao que está sendo consumido (Greenlaw, Shapiro e Taylor, 2017).

Segundo Krugman e Wells (2013), “o verdadeiro custo de algo é o seu custo de oportunidade”. Se alguém deseja um software, qual o verdadeiro custo disso? Isso envolve então o custo de oportunidade de uma segunda opção, por exemplo, não desenvolver o software e arcar com as consequências e investir o dinheiro.

Assim, novamente em software, o custo de oportunidade é um bom significado de valor. Perguntas que podem ser feitas, em caso de discussões de valor, estão ligadas alternativa de não fazer o produto, ou não corrigir um erro. E, no cálculo de um valor financeiro, o preço das outras ofertas que vão ser recebidas pelo cliente.

O conceito de valor na Economia, como tratado aqui, nos permite trabalhar com algumas ideias onde o homem é visto como um ser racional, que faz cálculos e toma a decisão de forma acertada baseada nesses cálculos. Porém o homem também é um animal emocional, e sua visão de valor envolve necessidades e desejos, como veremos na abordagem que o Marketing dá ao conceito de valor.

### 3.3. Valor em Marketing

O estudo de valor em marketing se inicia com o entendimento do que são **necessidades**, ou seja, “os requisitos básicos dos seres humanos”: água, comida, recreação, educação, etc. (Kotler e Keller, 2012). Necessidades são estados onde se sente uma privação (Kotler, Armstrong et al., 2017). Em uma empresa, comprar um software que emite nota fiscal segundo a legislação vigente é uma necessidade.

Necessidades se transformam em **desejos** “quando são direcionadas para objetivos específicos que podem satisfazê-las”(Kotler e Keller, 2012). Desejos “são a forma que as necessidades tomam quando moldadas pela cultura e personalidade individual”(Kotler, Armstrong et al., 2017).

Por exemplo, a fome gera uma necessidade de se alimentar, que pode se configurar como o desejo de um hambúrguer para um americano, ou um prato feito de arroz, feijão, bife e batata frita para um brasileiro.

Da mesma forma, dada uma necessidade de um tipo de software, uma organização pode desejar características específicas adicionais, como ser feita de código aberto.

As necessidades podem ser (Kotler e Keller, 2012):

- **declaradas**, que são ditas pelo cliente, mas não são necessariamente verdade;
- **reais**, que são o que o cliente realmente precisa;
- **não declaradas**, que são o que o cliente espera receber;
- **de algo mais (*delight*)**, que são o que o cliente gostaria, e
- **secretas**, que são como o cliente deseja ser visto pela sociedade.

Kotler e Keller (2012) ainda lembram que responder apenas a necessidade declarada pode não ser o bastante, e isso é verdade especialmente **em projetos de software, onde muitas vezes o cliente nem mesmo sabe suas necessidades reais**.

Uma **demand**a é um “desejo por um determinado produto, suportado por uma capacidade de pagar”(Kotler e Keller, 2012). Muitos querem lagosta no almoço, mas só podem pagar por algo mais barato.

Em software, a questão financeira é importante. É possível cotar um projeto que descrito em um parágrafo como algo entre 30 mil reais e quatro milhões de reais, dependendo da extensão do produto e da capacidade de operação necessária, como servidores e previsão de milhões de usuários.

Uma demanda pode ter oito estados (Kotler e Keller, 2012) em relação a um, ou mais, produtos ou serviços:

1. **demand negativa**, quando os consumidores não gostam do produto e podem até pagar para evitá-lo;
2. **demand não existente**, quando os consumidores não sabem que o produto existe ou não tem interesse no mesmo;
3. **demand latente**, quando os consumidores tem um necessidade grande que não é satisfeita por nenhum produto existente;

### 3. Valor

4. **demandas declinante**, quando os consumidores passam a comprar menos ou deixam de comprar um produto;
5. **demandas irregulares**, quando existe uma variação na compra do produto, em pequena escala de tempo (horas) ou grandes (estações do ano, meses);
6. **demandas plenas**, onde os consumidores estão comprando todos os produtos postos no mercado;
7. **demandas excessivas**, onde os consumidores gostariam de comprar mais produtos do que podem ser fornecidos, e
8. **demandas indesejadas**, onde os consumidores são atraídos por produtos com consequências sociais indesejadas.

Compreendendo essa sequência que começa com a necessidade, passa para o desejo e chega a demanda, é possível então falar de valor.

Para o marketing, o valor é a “a somatória dos benefícios tangíveis e intangíveis proporcionados pelo produto subtraída da somatória dos custos financeiros e emocionais envolvidos na aquisição desse produto”(Kotler e Keller, 2013). O valor é então uma “combinação de qualidade, serviço e preço”(Kotler e Keller, 2013). As percepções de valor “aumentam com a qualidade e o serviço, mas diminuem com o preço”(Kotler e Keller, 2013).

Segundo Drucker (1974) o valor, para um cliente, é a satisfação de um desejo. Um dos objetivos do marketing é aumentar esse valor, gerando satisfação no cliente. A **satisfação** é então um “julgamento comparativo de uma pessoa sobre o desempenho percebido de um produto em relação as expectativas”(Kotler e Keller, 2013).

#### 3.3.1. Os Elementos do Valor - B2C

Almquist, Senior e Bloch (2016) identificaram 30 elementos que fornecem valor para **consumidores**, i.e., pessoas, divididos em uma pirâmide<sup>1</sup> de quatro camadas<sup>2</sup>. Essa pirâmide é apresentada a seguir em forma de lista.

- mais alta* • **Impacto social**: auto-transcendência (ajudar outros ou a sociedade de forma mais ampla).
- **Mudança de vida**: fornecer esperança, auto-atualização, motivação, herança para as próximas gerações, afiliação/pertencimento.
  - **Emocional**: reduzir ansiedade, recompensa pessoal, nostalgia, design/estética, representação de status ou aspirações, bem estar, valor terapêutico, diversão/entretenimento, aumentar a atração pessoal, fornecer acesso a outros itens de valor.
- mais baixa* • **Funcional**: poupar tempo, simplificar, fazer dinheiro, reduzir risco, organizar, integrar aspectos diferentes da vida, conectar com outras pessoas, reduzir esforço, evitar problemas, reduzir custo, qualidade, variedade de escolha, apelo sensorial, informar.

---

<sup>1</sup>A pirâmide é uma construção usada para passar a ideia que os níveis inferiores são essenciais para os níveis superiores poderem ser alcançados

<sup>2</sup>Uma versão interativa pode ser acessada em <https://media.bain.com/elements-of-value/>, link válido em 9/2/2020

### 3.3. Valor em Marketing



Figura 3.1.: Elementos do Valor B2C. Fonte:Almquist, Senior e Bloch, 2016

Essa pirâmide pode ser comparada com a pirâmide de Maslow, apresentada na Figura 3.2, que mostra também as necessidades das pessoas em camadas cada vez mais abstratas. Sua aplicação é apropriada ao analisar valor em software que vai ser vendido para pessoas.

Os elementos podem ter importância diferente por pessoas e por indústria. Por exemplo, os elementos mais importantes para consumidores de *smartphones* são, em ordem decrescente (Almquist, Senior e Bloch, 2016): qualidade, reduzir esforço, variedade de escolha, organizar e conectar com outras pessoas<sup>3</sup>.

#### 3.3.2. Os Elementos do Valor - B2B

Mais tarde, Almquist, Cleghorn e Sherer (2018) desenvolveram modelo semelhante para negócios B2B, obtendo uma pirâmide um pouco mais complicada, com 40 elementos, onde há sub-níveis, representada na Figura 3.3 e na lista a seguir<sup>4</sup>. A importância de cada elemento novamente varia com a indústria. Essa segunda pirâmide é aplicável a software que vai ser vendido a organizações, porém leva em conta que as vendas são feitas para pessoas que participam das organizações. Os compromissos básicos são exigências para o vendedor.

##### *mais alta* • Valor inspiracional

- objetivo:

<sup>3</sup>Apesar de ser a missão original do *smartphone*, seu valor como conexão é só o quinto mais valorizado.

<sup>4</sup>Uma versão interativa pode ser acessada em <https://media.bain.com/b2b-eov/index.html>, link válido em 9/2/2020.

### 3. Valor

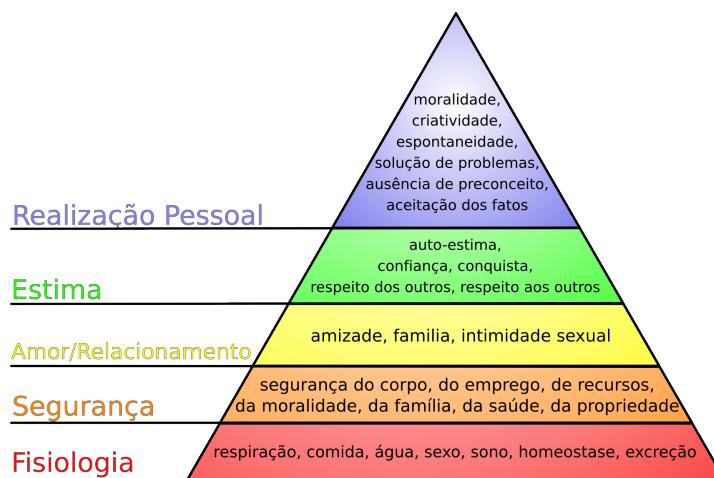


Figura 3.2.: Hierarquia das Necessidades de Maslow. Fonte: Wikipedia Commons por Felipe Sanchez (CC-BY-SA 3.0) e J. Finkelstein (GFDL)

- ◊ visão, ajuda o cliente a antecipar a direção do mercado;
- ◊ esperança, dá aos clientes e usuários esperanças no futuro da empresa, e
- ◊ responsabilidade social, ajuda o cliente a ser mais responsável socialmente.

- **Valor individual:**

- carreira:
  - ◊ expansão da rede (*network*), ajuda os usuários e colegas a expandir a rede profissional;
  - ◊ *marketability*, faz clientes e colegas mais *marketables* em seu campo, e
  - ◊ garantia de reputação, não atrapalha e pode aumentar a reputação do cliente no trabalho.
- pessoal:
  - ◊ design e estética, fornece produtos e serviços que são esteticamente agradáveis;
  - ◊ crescimento e desenvolvimento, ajuda usuários e colegas a crescer pessoalmente;
  - ◊ redução da ansiedade, ajuda clientes e outros na organização a se sentir mais seguros, e
  - ◊ diversão e vantagens, é agradável de interagir com ou dá recompensa de alguma forma.

- **Valor da facilidade de fazer negócios:**

- produtividade:
  - ◊ poupar tempo;
  - ◊ reduzir esforço;
  - ◊ diminuir problemas;
  - ◊ informação, e
  - ◊ transparéncia, fornece uma visão clara da organização do cliente.
- operacional:

### 3.3. Valor em Marketing

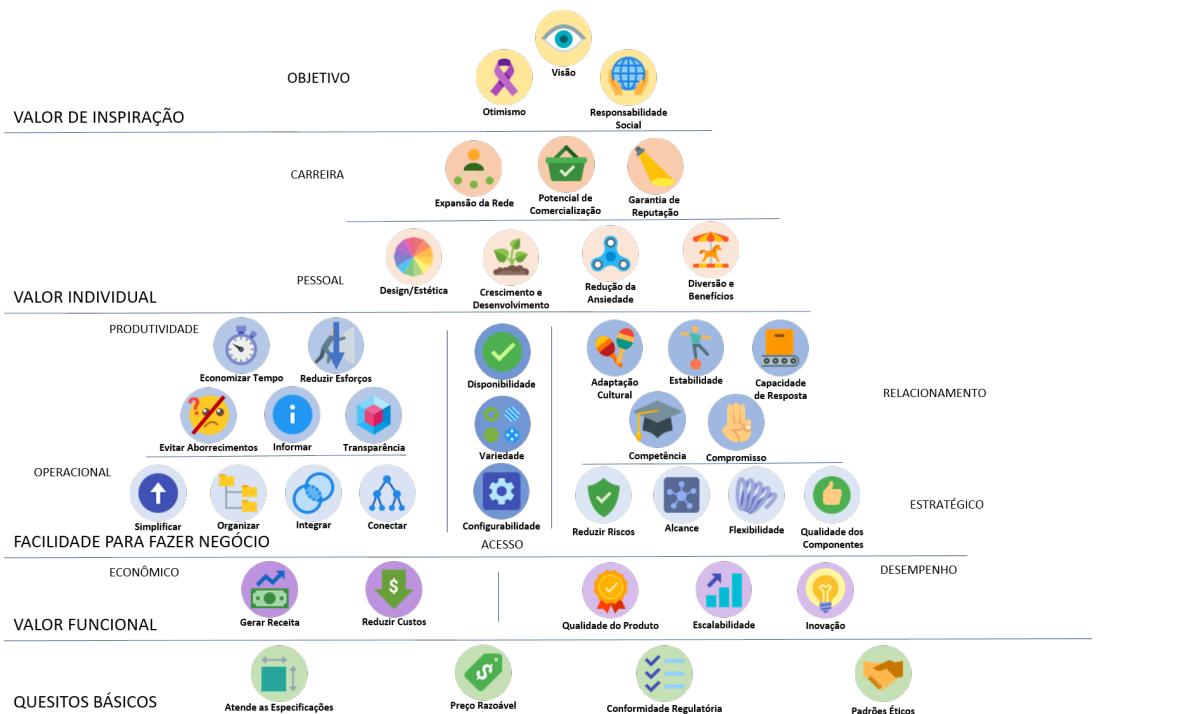


Figura 3.3.: Elementos do Valor B2B. Fonte: Almquist, Cleghorn e Sherer, 2018

- ◊ organização;
- ◊ simplificação, reduz complexidade;
- ◊ conexão, conecta organização e usuários com outros interna e externamente, e
- ◊ integração, ajuda o cliente a integrar diversas facetas do negócio.
- acesso:
  - ◊ disponibilidade, garante que o bem ou serviço está disponível quando e onde necessário;
  - ◊ variedade, fornece uma variedade para escolha, e
  - ◊ configurabilidade, permite configurar o bem ou serviço de acordo com as necessidades do cliente.
- relacionamento:
  - ◊ responsividade, responde rápida e profissionalmente as necessidades da organização;
  - ◊ perícia, fornece *know-how* para o mercado ou indústria relevante;
  - ◊ compromisso, mostra que está compromissado com o sucesso do cliente;
  - ◊ estabilidade, é uma empresa(bem ou produto) estável no futuro previsível, e
  - ◊ ajuste cultural, se encaixa na cultura do cliente.
- estratégico:
  - ◊ redução de risco, protegendo o cliente;
  - ◊ alcance, permite o cliente operar em mais locais ou segmentos do mercado;

### 3. Valor

- ◊ flexibilidade, vai além dos bens ou padrões comuns para permitir customização, e
- ◊ qualidade de componentes, melhora a qualidade percebida dos produtos e serviços do cliente.

- **Valor funcional:**

- econômico:
  - ◊ redução de custos, e
  - ◊ aumento de receitas.
- desempenho:
  - ◊ qualidade do produto;
  - ◊ escalabilidade, e
  - ◊ inovação.

mais baixa • **Compromissos básicos (*table stakes*<sup>5</sup>)**

- satisfazer especificações;
- preço aceitável;
- conformidade regulatória, e
- padrões éticos.

Ambas as pirâmides tentam ser exaustivas, mas foram criadas a partir de pesquisas de campo, o que quer dizer que podem existir outros elementos não detectados, porém isso não é esperado em geral.

Além disso, elas devem ser usadas em função dos elementos mais desejados em uma indústria ou cliente específico. Por exemplo, investigando a contribuição proporcional dos elementos para os clientes de infraestrutura de TI, com uma soma de 100%, (Almquist, Cleghorn e Sherer, 2018) encontraram que os três principais elementos são: qualidade do produto (7,8%), perícia (6,1%) e responsividade (5,5%). Poupar tempo, por exemplo, era décimo elemento na ordem e com proporção de apenas 3,0%. Na prática, a importância de cada elemento pode ser levantada com cada cliente.

#### 3.3.3. Outras Técnicas

Tendo em vista que a finalidade do Marketing é engajar e gerenciar relações lucrativas com os clientes, é razoável que parte do trabalho de Marketing seja identificar o que é valor para o cliente, de forma que a organização possa atendê-lo com qualidade. Assim, muitas técnicas de Marketing se dedicam a isso e está além deste livro cobrir todas.

Outras técnicas interessantes provenientes do Marketing são o QFD (Franceschini, 2016), adequada tanto a software sob encomenda quanto a produtos a serem lançados no mercado, e a Estratégia do Oceano Azul (Kim e Mauborgne, 2005), adequada apenas a novos produtos a serem lançados no mercado.

---

<sup>5</sup>No linguajar de negócios em inglês, *table stakes* é o mínimo que você deve fazer para um mercado ou negócio específico.

## 3.4. Priorização Baseada em Valor para o Cliente

Em muito projetos é mais prático colocar os itens a serem realizados em uma lista priorizada do que dar um valor para cada item, sejam eles demandas do cliente, requisitos em português, casos de uso, histórias do usuário ou outros métodos.

A priorização é um processo para “determinar a importância relativa de informações”(IIBA, 2011). Ela permite várias abordagens:

- agrupamento, normalmente em poucas classes pré-determinadas;
- *ranking* ou ordenação;
- orçamento ou *time-boxing*, por meio da alocação de recursos finitos e
- negociação, em busca de um consenso.

### 3.4.1. Técnicas simples de ordenação

Algumas das técnicas que podem ser usadas para criar uma estimativa relativa de valor com clientes e outras partes interessadas são baseadas em criar uma pontuação ou separar os itens que têm o valor estimado em grupos de níveis de prioridade diferentes(Satpathy et al., 2016).

Métodos simples de priorização incluem:

- esquemas simples de pontuação, onde cada item do projeto analisado recebe um número entre uma variação pequena, como de 1 a 3 ou 1 a 5, ou uma rótulo como “Alto”, “Médio” ou “Baixo”, criando grupos ordenados;
- **dinheiro de brinquedo** ou **100-pontos**, onde é dada uma quantidade de dinheiro de brincadeira<sup>6</sup>, como 100 moedas, ou pontos, para o cliente associar aos itens do projeto, algumas vezes submetido a algumas regras, como não ter X itens com o mesmo valor (Leffingwell e Widrig, 1999), criando uma ordem parcial baseada na alocação de recursos, e
- ordenação de cartões contendo um item cada (J. Robertson e S. Robertson, 1998), mas também realizada em planilhas eletrônicas, ou em softwares como *Trello*.

### 3.4.2. Método MoSCoW

Na priorização **MoSCoW**, cada item é associado a um conceito entre “*Must Have*”, deve ter, “*Should Have*”, deveria ter, “*Could Have*”, poderia ter e “*Won’t Have*”, não irá ter (IIBA, 2011, p. 108), criando grupos.

Há uma diferença clara entre esse método e separar em grupos que determinam um grau de prioridade: no método MoSCoW os grupos tem um significado específico.

---

<sup>6</sup>Facilmente obtida em uma caixa de Banco Imobiliário

### 3. Valor

Entre os “*Must Have*” devem estar apenas os itens essenciais ao projeto, ou seja, aqueles que, não existindo, colocam o sucesso do projeto em risco. São fatores críticos do sucesso do projeto.

Nos dois grupos seguintes, “*Should Have*” e “*Could Have*”, cai a necessidade. Espera-se que contenham funcionalidades desejadas, mas que não sejam essenciais ao sucesso. No caso do “*Could Have*”, poderiam ser apenas melhorias eventuais ou mesmo o que é conhecido como *gold plating*, isto é, decoração que não soma realmente valor.

Finalmente os “*Won’t Have*” mostram requisitos que foram levantados, mas não são necessários ou mesmo podem causar problemas ao projeto.

Ao longo do tempo, principalmente em projetos iterativos, como os projetos ágeis, é comum um item mudar de grupo de uma iteração para outra. Por exemplo, quando todos os itens “*Must Have*” estão realizados, pode acontecer de outros itens passar a ser “*Must Have*”. Mesmo itens que são indesejados (“*Won’t Have*”) em algum momento do projeto podem se tornar necessários, e o inverso também é verdadeiro.

Um exemplo de item indesejado que pode se tornar desejado em um projeto é a compra de um insumo ou equipamento muito caro, que passa a se tornar necessário em um certo ponto do projeto. Isso pode acontecer se a alternativa não atende às expectativas ou se o problema se tornou mais difícil do que planejado.

O método MoSCoW não é exclusivo da área de software, mas é parte importante de métodos de gestão de projetos e foi particularmente utilizado no método DSDM (Stapleton, 2003).

#### 3.4.3. Análise de Kano

O modelo de Kano é um método para avaliar a reação emocional de clientes a características individuais de um produto. Kano detectou cinco respostas emocionais a essas características (Moorman, 2012):

- encantadores ou atrativos (*delighters*), “eu gosto disso”, que trazem satisfação se presentes, mas não trazem insatisfação se não estiverem presentes, normalmente porque são inesperadas e endereçam necessidades não reconhecidas;
- unidimensionais (*satisfiers*), que causam satisfação se estiverem presentes e insatisfação se não estiverem presentes;
- obrigatórios ou esperados (*dissatisfiers*), que causam insatisfação se não estiverem presentes, mas não causam satisfação se estiverem presentes;
- indiferentes (*indifferent*), ou
- indesejados.

Para classificar em uma dessas cinco respostas emocionais, são feitas duas perguntas ao cliente (Moorman, 2012):

### 3.4. Priorização Baseada em Valor para o Cliente

- Como você vai se sentir se a funcionalidade estiver presente?
- Como você vai se sentir se a funcionalidade não estiver presente?

As respostas devem ser em uma escala ordinal:

- eu gosto disso;
- eu espero isso;
- eu sou neutro em relação a isso;
- eu posso tolerar isso, ou
- eu não gosto disso.

Finalmente, a classificação é feita cruzando as respostas na Tabela 3.1.

Tabela 3.1.: Resolvendo a classe de Kano por meio do cruzamento de duas perguntas  
(Moorman, 2012)

		Questão Negativa				
Questão Positiva	Gosto	Gosto	Espero	Neutro	Tolero	Não Gosto
	Gosto	Conflito!	Atrativo	Atrativo	Atrativo	Unidimensional
	Espero	Indesejado	Indiferente	Indiferente	Indiferente	Obrigatório
	Neutro	Indesejado	Indiferente	Indiferente	Indiferente	Obrigatório
	Tolero	Indesejado	Indiferente	Indiferente	Indiferente	Obrigatório
	Não Gosto	Indesejado	Indesejado	Indesejado	Indesejado	Conflito!

As categorias de Kano não são permanentes, e as características de um produto, ou tipo de produto, vão migrando de atrativas, para unidimensionais, e para obrigatórias, e podem até mesmo se tornar indesejadas. O tamanho do celular, por exemplo, sofreu essa mudança com o tempo.

J. Robertson e S. Robertson (2006) usam o método de Kano no seu método Volare. Nesse método o valor para o cliente é definido em duas perguntas: quão satisfeito você se sentirá se esse requisito for implementado e quão insatisfeito você se sentirá se esse requisito não for implementado. Segundo o casal, os graus de satisfação e insatisfação são a melhor mecanismo para saber que requisitos o cliente considera com maior valor. J. Robertson e S. Robertson, porém, não se referem a Kano, mas a um autor que descreve suas técnicas. As notas dadas também são de 1 a 5.

O modelo de priorização do método Volare permite criar um gráfico de quatro quadrantes, como o da Figura 3.4. O mesmo princípio pode ser aplicado com as respostas do método de Kano.

#### 3.4.4. Prioridade e Consenso

Todas as técnicas explicadas nesta seção foram descritas como se o cliente tivesse uma opinião única, porém isto não é sempre verdade. Como projetos possuem muitas partes interessadas, cada parte interessada pode trazer uma visão diferente da realidade.

### 3. Valor



Figura 3.4.

Para isso existem técnicas que visam atingir o consenso. A questão é que algumas técnicas atingem um ponto que não é considerado realmente consenso. No sentido estrito, consenso significa que todos concordaram com algo, porém algumas técnicas tentam resolver problemas de opiniões diferentes com votação, sendo possível ser uma votação por maioria simples, ou seja, metade dos votos mais um, ou por maioria qualificada, onde se exigem mais votos, como dois terços dos votos. Outras formas de eleição podem ser mais relacionadas com o conceito de consenso, como o voto em múltiplas rodadas, onde itens menos votados são eliminados em cada rodada. O voto em dois turnos é a versão mais simples desse método. Métodos de votação preferencial tentam simular as múltiplas rodadas. Nesses métodos o eleitor ordena os candidatos de acordo com sua preferência, eliminando o candidato com menor preferência e recontando os votos, agora sem esse candidato, até que um candidato se eleja por maioria da primeira preferência não eliminada de cada eleitor (Adiel Teixeira de Almeida, 2019; Amy, 2000).

Algumas técnicas confundem o consenso com o consentimento. Consentir é concordar que a decisão foi tomada da melhor forma possível, mas não necessariamente concordar com ela.

Provavelmente a técnica mais famosa de atingir o consenso é o método de Delphi (Dalkey, 1967). Na sua versão original, eram usadas várias rodadas, feitas pelo correio ou meio a distância, onde especialistas deveriam avaliar uma afirmação dentro de um espectro linear, por exemplo, entre “não vai acontecer” e “muito provável”, ou entre uma data inicial e final de quando um evento era esperado. A cada rodada uma pessoa devia coletar todos os votos, fazer uma estatística, por exemplo mostrando a mediana e os quartis, e suas justificativas, e reenviar essa agregação para que todos os especialistas, a partir dos novos argumentos, refizessem sua avaliação. O Método de Delphi tem três características importantes: as opiniões são anônimas, o feedback é controlado e a resposta do grupo é estatística. A anonimização é importante para eliminar o efeito de dominância social

### 3.4. Priorização Baseada em Valor para o Cliente

de algumas pessoas, pela experiência ou por sua retórica poderosa (Helmer-Hirschberg, 1967). Existem variações do método de Delphi com objetivo específico, como o *Planning Poker*- (Cohn, 2005, p 56).

Também existem técnicas que permitem calcular o grau de consenso (Meskanen e Nurmi, 2006). Isso pode ser necessário, por exemplo, quando cada pessoa escolhe uma ordenação diferente. Nesse caso, havendo um valor específico que indique o consenso, é possível negociar até que ponto a discussão precisa continuar. Isso pode variar com o tipo de decisão, como por exemplo a discussão entre uma ordenação ou a escolha do melhor. Ross (2017, pags. 265–321) fornece métodos para ordenar e calcular diferentes tipos de consenso a partir de comparações par a par de preferência, mesmo que não sejam transitivas.

O **dot-voting** (Gray, Brown e Macanufo, 2010) é uma técnica típica do Design Thinking e dos métodos ágeis. Nessa técnica, os itens a serem priorizados são colocados em cartões em um quadro e cada participante recebe um número de votos, sendo 3 um número padrão na literatura, normalmente na forma de adesivos em forma de círculo (*dots*). Os participantes então colam seus adesivos, isto é, dão votos não identificados, nos cartões.

A técnica é muito usada, mas alguns defeitos já foram reconhecidos. Um deles é o efeito em cascata, um item já votado tende a chamar atenção e receber mais votos. Outro é que é comum o excesso de escolhas. Além disso, se os itens estiverem em níveis diferentes de uma hierarquia, é possível que um item menos desejado seja escolhido, porque o item mais desejado estava representado em duas possibilidades. Existem soluções, e várias delas vêm do estudo do comportamento de eleitores em diferentes formas de eleições.

#### 3.4.5. Usando mais de uma dimensão

Os métodos anteriores usavam apenas uma dimensão para determinar a prioridade, porém é possível utilizar outras dimensões. Um exemplo é usar uma dimensão que determine o benefício e outra que determine o custo, já que duas dimensões nos permitem novamente usar um diagrama 2D que pode ser dividido em quadrantes, sendo fácil de analisar. A dimensão que determina o benefício é sempre feita em função das partes interessadas principais, os clientes. Já a dimensão que determina o custo é melhor avaliada pelo equipe que executará o projeto, como a equipe de desenvolvimento. Nesse caso, é necessário utilizar técnicas de predição, como o Planning Poker Cohn, 2004 ou Pontos de Função (IFPUG, 2010).

Definido o benefício e o custo, é possível analisar o resultado em um diagrama com 4 quadrantes, como o da Figura X.

Dada uma tabela com itens, onde tanto o benefício quanto o custo receberam um valor de 1 a 5, os itens podem ser posicionados na gráfico e uma ordem de prioridade pode ser determinada. 3.2. A Tabela 3.2 permite gerar o gráfico da Figura 3.6, onde as setas mostram uma sequência possível de priorização, AFHIEBJG, onde os itens C e D

### 3. Valor



Figura 3.5.: Gráfico com quadrantes para comparar prioridades usando benefício e custo.

foram desprezados. Outra sequência possível seria AFHIEDJGC. Com essa visualização é possível melhorar a expectativa do cliente quanto a prazos de atendimento das suas reais necessidades.

Tabela 3.2.: Benefício e esforço para itens de um project backlog imaginário.

Funcionalidade	Benefício	Esforço
A	5	1
B	3	2
C	1	3
D	2	4
E	4	3
F	5	2
G	1	2
H	5	3
I	5	4
J	1	1

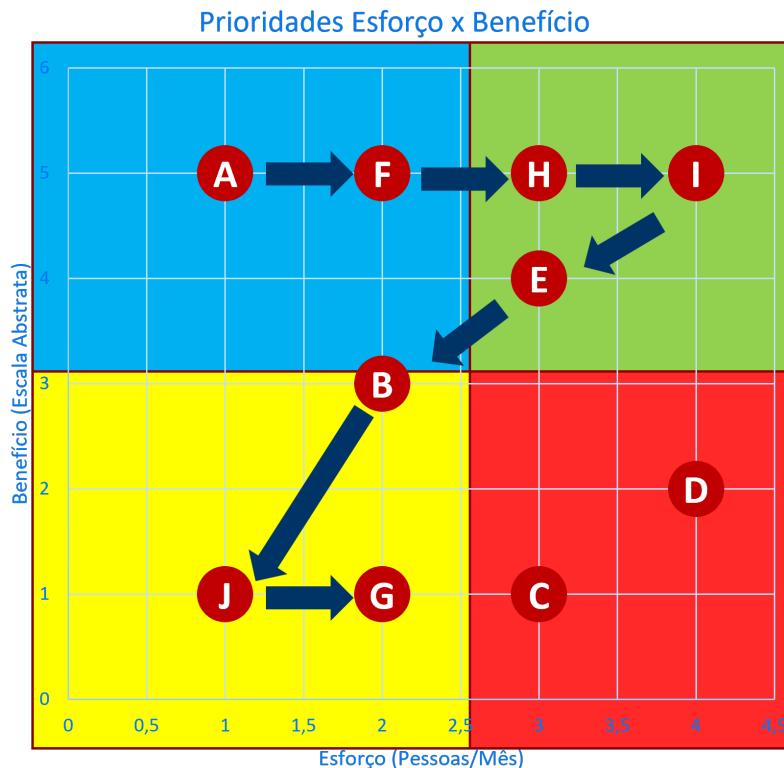


Figura 3.6.: Gráfico com quadrantes, determinados arbitrariamente pelo usuário, para os itens de projeto listados na Tabela 3.2. As setas mostram uma possível sequência de prioridade. Os itens C e D foram considerados desnecessários ao projeto.

### 3.4.6. Prioridade e Dependências

Suponha que um item A seja mais prioritário que um item B, porém A não possa ser usado sem que B esteja pronto. O que fazer?

Esse é um acontecimento comum, e mostra que há uma diferença entre as prioridades do negócio e as prioridades de um projeto que busca atender o negócio. Nesse caso, podem ser investigadas as seguintes soluções:

1. Alterar a forma como o software é desenvolvido, de forma que A funcione sem que B esteja pronto;
2. Dividir B em pedaços, de forma que apenas parte de B seja necessária para A funcionar;
3. Aceitar que B tem que ser feito antes de A.

### 3.5. Valor Financeiro de Projetos

Apesar de não ser o objetivo principal desse capítulo, é interessante observar as formas de calcular o valor financeiro de um projeto, ou subprojeto.

Antes de tratar das estimativas financeiras usadas em projetos, é importante conhecer os seguintes conceitos, como definidos por Dal Zot e Castro (2015):

- **principal** (P) é o capital inicial de um empréstimo ou uma aplicação financeira, também conhecido como **valor presente** (VP), **valor atual** (VA), **valor descontado**, ou por seu nome em inglês, ***present value*** (PV);
- **juro** (J) é a **remuneração** do capital emprestado, da parte de quem paga é uma despesa ou custo financeiro, da parte de quem receber é um rendimento ou renda financeira;
- **montante** (S) é o **saldo**, ou **valor futuro** (VF), ou ***future value*** (FV), de um empréstimo ou de uma aplicação financeira. Pode ser também chamado de **valor de resgate**, ver equação 3.1.
- **prazo** (n) é o período de tempo que dura o empréstimo ou a aplicação financeira, podendo ser medido em dias, meses, anos...
- **prestaçāo** ou **pagamento** (P,PGTO,PMT) se refere ao valor de pagamentos quando feitos em um número maior do que 1.
- **taxa de juros** (i) , ou simplesmente **taxa**, é o quociente entre juros e o principal, ver equação 3.2;

$$S = P + J \quad (3.1)$$

$$i = \frac{J}{P} \quad (3.2)$$

Analizando essas definições podemos detectar uma propriedade importante do dinheiro e que é a base da Matemática Financeira: o dinheiro muda de valor ao longo do tempo. Normalmente, como os cenários de inflação são muito mais comuns que os de deflação, R\$100,00 hoje compram mais que compraram daqui a um ano e menos do que compravam um ano atrás. Por meio de cálculos como o do Valor Presente, como veremos no caso do Valor Presente Líquido, podemos comparar opções de investimentos ou empréstimos.

Calculadores financeiros, como a HP-12C e o Excel permitem calcular facilmente o Valor Presente e o Valor Futuro com pagamentos, ou retiradas, fixas, em um período determinado, ao com uma taxa de juros fixas.

Por exemplo, se você tiver R\$10.000,00 por mês e puder aplicar em um projeto, deve procurar uma boa alternativa de investimento para poder saber se, financeiramente, vale a pena usar o dinheiro no projeto, ou se é melhor usar esse investimento. Com a fórmula do Valor Futuro (VF em português ou FV em inglês) esse cálculo pode ser facilmente feito no Excel, como mostrado na Figura 3.7.

Investimento Alternativo	
Taxa de Juros	1,00%
Período	12
Pagamento	R\$ 10.000,00
Valor Presente	R\$ 0,00
Valor Futuro	R\$ 126.825,03

(a) Valores e formatação

Investimento Alternativo	
Taxa de Juros	0,01
Períodos	12
Pagamento	10000
Valor Presente	0
Valor Futuro	=FV(Taxa_de_Juros;Períodos;Pagamento;Valor_Presente;0)*-1

(b) Fórmulas

Figura 3.7.: Cálculo do valor futuro de um investimento de R\$10.000,00 por mês com uma taxa de juros de 1%.

Existem várias práticas que são utilizadas para calcular o valor financeiro de um projeto, entre elas (Satpathy et al., 2016):

- Custo Total de Propriedade (TCO - *Total Cost of Ownership*), que calcula todo custo de um serviço ou produto, incluindo a realização e a o uso do mesmo.
- Retorno do Investimento (ROI - *Return on Investment*), o método mais usado que é basicamente o lucro sobre o custo;
- Valor Presente Líquido - VPL (NPV - *Net Peesent Value*), onde se calcula a diferença entre a receita do projeto e seus custo ao longo do tempo, corrigido para o tempo presente, e
- Taxa Interna de Retorno - TIR (IRR - *Internal Rate of Return*) é a taxa de desconto de um investimento que torna o NPV zero para todos os fluxos de caixa de um projeto, não podendo ser calculado analiticamente, quanto maior for, mais desejável é o projeto (Hayes, 2019).

Devemos ter cuidado com projetos com resultados subjetivos, que não podem ser medidos diretamente pelo valor financeiro. Essa questão não será discutida nesse livro.

### 3.5.1. Custo Total de Propriedade

Quando se analisa o custo de um sistema é normal falar de Custo Total de Propriedade, conhecido pela sigla em inglês TCO (Total Cost of Ownership). O TCO é o valor presente de todos os custos a serem feitos durante a vida de um sistema, produto, serviço ou equipamento(Anklesaria, 2008).

### 3. Valor

Tabela 3.3.: Valores usados para calcular o TCO de uma impressora.

Impressora	Laser A	Ink A	Laser B
Preço	R\$ 880,00	R\$ 400,00	R\$ 2.100,00
Preço Toner	R\$ 480,00	R\$ 105,00	R\$ 539,00
Páginas Toner	1000	480	12.000

Por exemplo, se decidirmos trocar todo o parque computacional de uma empresa que usa Windows para Linux, mesmo que o custo do Linux seja zero, o TCO é bem alto, pois envolve o processo de troca, novos profissionais, treinamento, etc... Outro exemplo comum é o da compra de uma impressora. Seu TCO não envolve apenas o custo da impressora, mas também o custo do material consumível, quando uma certa produção é prevista. Por isso é que grandes empresas compram menos impressoras, porém impressoras maiores e mais caras, para baixar o TCO, já que o custo por impressão delas acaba saindo menor.

Para o software desenvolvido vale o mesmo conceito. Qual seu TCO? Envolve o preço pago pelo software mais tudo que vai ser pago para possibilitar a implantação e utilização do produto (instalação, cursos de treinamento, manutenção mensal, etc...). Espera-se, em uma decisão racional, que o TCO seja menor que o benefício trazido pelo sistema.

Um exemplo típico é a análise a ser feita na compra de uma impressora. Impressoras mais caras normalmente tem o preço por página impressa mais barato, porém só a partir de um ponto de uma quantidade grande de impressões.

A Tabela 3.3 e a Figura 3.8 mostram como isso acontece. Nelas vemos qual a melhor alternativa, considerando apenas o preço da impressora e do tipo de impressão, qual a impressão mais barata. É possível, por exemplo, somar o gasto de energia a essa conta.

Anklesaria (2008) divide os custos em possíveis classes:

- **preço de compra**, e
- custos internos, que se dividem em
  - **custos de aquisição**, como transporte, inspeção, armazenagem;
  - **custos de uso**, como manutenção, garantias, e
  - **custos do fim de vida**, como custos para desmontar um projeto, etc.

#### 3.5.2. Retorno do Investimento

O cálculo do ROI depende de haver uma previsão para a receita corrente  $R$  e para o custo  $C$  do projeto. Sua fórmula é:

$$ROI = \frac{R - C}{C} \tag{3.3}$$

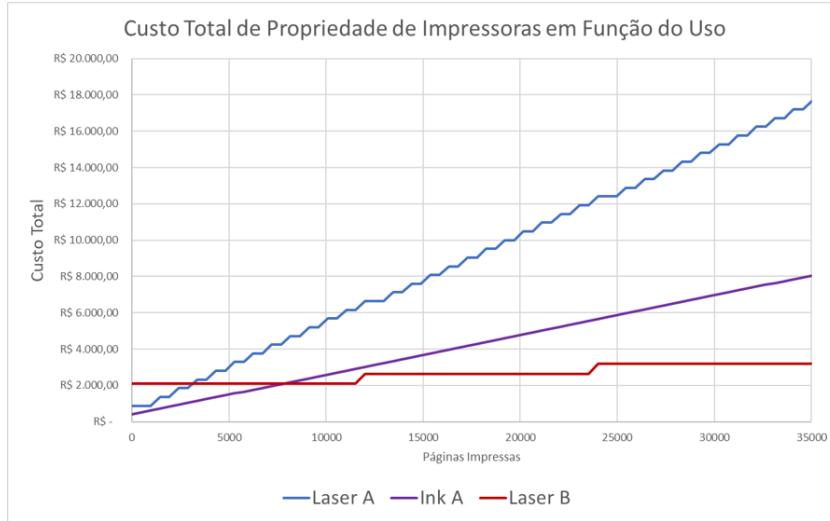


Figura 3.8.: Gráfico do TCO de impressoras em função da quantidade de páginas impressas. Percebe-se que após aproximadamente 3500 páginas é melhor comprar uma impressora a laser mais cara, caso esse fosse o único fator de análise.

O cálculo do ROI de um projeto que está previsto custar R\$ 500.000,00 e pretende obter como benefício um valor de R\$ 700.000,00 é:

$$ROI = \frac{700 - 500}{500} = \frac{200}{500} = 40\% \quad (3.4)$$

O ROI pode ser usado para comparações com taxas que o mercado financeiro paga por um investimento do mesmo valor. Assim, só valeria a pena fazer um projeto, financeiramente falando, se ele resultasse em um retorno mais alto do que poderia ser obtido com o investimento em algum título do mercado.

Um cuidado ao analisar o ROI de um projeto é o fato de ser um valor percentual. Nesse caso, um projeto de R\$ 10.000,00 com um ROI de 150% pode parecer mais interessante que um projeto de R\$ 1.000.000,00 com um ROI de 1%, porém o segundo gera muito mais dinheiro que o primeiro.

### 3.5.3. Valor Presente Líquido

O Valor Presente Líquido é o valor atualizado para o presente de todos os gastos e ganhos de um projeto. Essa atualização é feita com uma taxa de desconto fixa. Isso significa que se você ganhar R\$ 100,00 daqui a um ano, e a inflação for de 5% ao ano, hoje isso tem o valor equivalente ao valor presente que, quando somado de 5%, resultasse em R\$100,00. A conta é  $1/1,05$ , que é aproximadamente R\$95,24.

### 3. Valor

A fórmula do VPL é(Kenton, 2019):

$$VPL = \sum_{t=1}^T \frac{C_t}{(1 + i)^t} \quad (3.5)$$

onde  $C_t$  é o caixa líquido, receitas menos despesas, em um período  $t$  e  $i$  a taxa de desconto em um investimento alternativo, sendo  $T$  o número de períodos.

Esse cálculo é exemplificado na Tabela 3.4, onde usamos o método de calcular o saldo de cada período e atualizá-lo para o presente, com uma taxa fixa de 5%.

Período	Custo	Receita	Caixa	Valor Presente
0	R\$ 50.000,00	R\$ 0,00	-R\$ 50.000,00	-R\$ 50.000,00
1	R\$ 40.000,00	R\$ 52.500,00	R\$ 12.500,00	R\$ 11.904,76
2	R\$ 40.000,00	R\$ 52.000,00	R\$ 12.000,00	R\$ 10.884,35
3	R\$ 50.000,00	R\$ 61.500,00	R\$ 11.500,00	R\$ 9.934,13
4	R\$ 20.000,00	R\$ 31.000,00	R\$ 11.000,00	R\$ 9.049,73
5	R\$ 20.000,00	R\$ 30.500,00	R\$ 10.500,00	R\$ 8.227,02
6	R\$ 20.000,00	R\$ 30.000,00	R\$ 10.000,00	R\$ 7.462,15
7	R\$ 20.000,00	R\$ 29.500,00	R\$ 9.500,00	R\$ 6.751,47
<b>Total</b>				<b>R\$14.213,63</b>

Tabela 3.4.: Tabela de cálculo do Valor Presente Líquido, considerando saldos diferentes em cada período. Foi usada uma taxa de 5% a cada período. A fórmula para cada Valor Presente é Caixa<sub>i</sub>/(1 + Taxa)<sup>Período</sup>

#### 3.5.4. Taxa Interna de Retorno

A Taxa Interna de Retorno é um cálculo um pouco mais complicado, exigindo um processamento de tentativa e erro. Ela indica de quanto deveria ser a taxa de desconto para o VPL ser zero, isto é, o resultado do projeto não ser nem positivo, nem negativo. O processo de cálcula-la é numérico, mas o Excel e calculadoras financeiras fornecem funções para isso.

A fórmula que relaciona o VPL com o TIR é(Hayes, 2019):

$$0 = VPL = \sum_{t=1}^T \frac{C_t}{(1 + TIR)^t} - C_0 \quad (3.6)$$

onde  $C_0$  é o investimento inicial total.

Quanto mais alto for o TIR, melhor será o resultado do projeto, sempre lembrando que como o cálculo é percentual, isso depende do valor do projeto. Uma maneira de entender isso é que você precisaria de uma aplicação melhor que a taxa para o projeto não valer a pena financeiramente.

## 3.6. Valor em Software

Conhecendo algumas definições na Economia e no Marketing, podemos compreender melhor outras definições que encontramos na Engenharia de Software.

Segundo Erdogmus, Favaro e Halling (2006) , **valor** é a “diferença entre benefícios e custos de um bem, ajustados ao risco, em um certo momento de tempo”.

Ele é dirigido por valores individuais ou coletivos. As partes interessadas esperam obter algum **benefício**, seja ele tangível ou intangível, econômico ou social, monetário ou utilitário, ou ainda estético ou ético (Biffl et al., 2006).

Valor significa esse benefício final, que existe geralmente nos olhos do observado e admite múltiplas caracterizações (Biffl et al., 2006).

O risco, que ainda não tínhamos encontrado em outras definições, é importante, e é parte das técnicas de gestão de projeto modernas considerá-lo. O risco pode ser incluído inclusive indiretamente no custo total previsto de um projeto multiplicando sua probabilidade pelo seu impacto. Isso faz pouco sentido para um risco único, mas já faz sentido para o conjunto total de riscos de um projeto.

Assim, o valor de algo pode ser visto como uma fórmula matemática, de uma forma básica como:

$$\text{benefício} - \text{custos} \quad (3.7)$$

Ou, considerando os riscos:

$$(\text{benefício} - \text{custos}) \times \text{correção por risco} \quad (3.8)$$

A partir dessa interpretação, busca valor é buscar os maiores benefícios a um custo que valha a pena, com risco baixo. Esses benefícios devem ser identificados no início do projeto, e estar alinhados com o planejamento estratégico da empresa.

É importante frisar que o valor deve ser em relação a organização, não ao software propriamente dito. Assim, algumas funcionalidades que poderiam tornar o software muito mais poderoso podem, na prática, não ser úteis, não trazer benefícios, custar caro demais ou ter risco muito alto. E isso depende de cada projeto, ou cada aplicação do software.

Gane e Sarson (1979) introduziram um o acrônimo **IRACIS**, que identifica três formas de agregar valor (Gane e Sarson, 1979; Ruble, 1997):

- Aumentar Faturamento (*Improve Revenue*);
- Evitar Custos (*Avoid Costs*), e
- Melhorar Serviços (*Improve Services*).

### 3. Valor

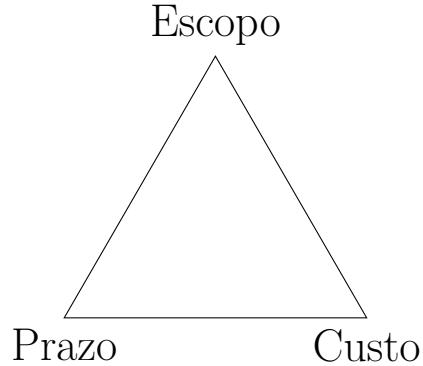


Figura 3.9.: O triângulo do valor para a maioria dos projetos.

#### 3.6.1. Triângulo do Valor

É comum se falar, em projetos de software, que existem três parâmetros, que são qualidade (ou escopo), tempo e custo, formando um triângulo, mas o cliente só pode definir dois. Apesar de ser tratado como um brincadeira, é realmente impossível arbitrar esses três valores de forma independente. Ou seja, a afirmação “eu quero um software com a funcionalidade X em tempo Y e custo Z” não pode ser feita sem conhecimento de causa (BeSeen, 2015; Kerzner, 2017).

A princípio, isso pode ser entendido a partir do fato que Barry W. Boehm et al. (2000) mostraram que há uma relação entre a funcionalidade e qualidade desejada, e o esforço necessário para desenvolver o sistema, o que é óbvio. O que não era tão óbvio é que, além disso, dado um esforço, o tempo necessário para desenvolver o sistema também é determinado. Podem ser feitas certas modificações na previsão estatística, porém se forem feitas no sentido de diminuir o prazo necessário para a entrega do produto, elas implicam em risco maior. O assunto já foi tratado há muito anos por Brooks (1978) em seu livro adequadamente chamado *The Mythical Man-Month*.

Kerzner (2017) trata escopo, tempo e custo como restrições primárias que competem entre si em um projeto tradicional, e ainda apresenta restrições secundárias: reputação e imagem, risco, qualidade e valor. Já em projetos complexos, ainda segundo Kerzner, as restrições primárias passam a ser reputação e imagem, valor e qualidade, enquanto escopo, risco, custo e tempo passam a ser restrições secundárias. Podem aparecer também outros fatores, como segurança e estética, que aparece em parques de diversão, por exemplo.

#### 3.6.2. Valor em Métodos Ágeis

É comum que métodos ágeis, como Scrum (Satpathy et al., 2016), proponham priorizar, de forma contínua, em ciclos curtos, os requisitos baseados no valor de negócio que é entregue aos clientes e usuários. O foco do Scrum, por exemplo, é tornar entregar valor, na forma de software executável, rapidamente, por meio de entregues incrementais a

cada ciclo. Não há, porém, uma definição específica do significado da palavra valor, o que leva a necessidade de um entendimento maior do termo. Mesmo assim, cada história do usuário representa um valor a ser entregue ao cliente.

## 3.7. Conclusão

Não existe uma ideia unificada do que é valor, principalmente quando o conceito é comparado entre áreas distintas.

Como os clientes do analista de sistemas, partes interessadas nos projetos, estão em várias áreas diferentes, mesmo em um só projeto, então podem interpretar valor de formas diferentes também.

Mesmo assim, a **principal função do desenvolvimento de software é produzir um software que tenha valor ao usuário**, seja esse valor calculado por uma fórmula ou um conceito mais qualitativo.

Para isso é importante definir o que é valor no início de qualquer projeto, de maneira consensual entre as partes interessadas. Isso é uma prática ágil que deve sempre ser adotada. Esse valor pode ser considerado de forma absoluta ou relativa.

A técnica MoSCoW (IIBA, 2011, p. 108), que dá valor de forma relativa, também é muito prática e pode ser associada a qualquer outra. A partir dela pode ser mais fácil fazer uma ordenação de uma grande lista de requisitos, por exemplo.

Já a técnica de Kano (Moorman, 2012; Sr., 2015), ou seu uso como proposto pelo Volare (J. Robertson e S. Robertson, 2006), permite trabalhar sobre a importância de cada item. Há, na verdade, uma relação entre o resultado da técnica de Kano e os grupos do método MoSCoW.

O uso dos Elementos de Valor (Almquist, Cleghorn e Sherer, 2018; Almquist, Senior e Bloch, 2016) pode ser uma solução adequada para o processo de discussão com as partes interessadas em busca de definições mais precisas e completas do verdadeiro valor.

## 3.8. Exercícios

### Exercício 3.1:

Procure outras definições de valor na Economia ou investigue mais sobre as definições dadas. Discuta a diferença entre essas definições e como elas mostram a evolução do entendimento do que é valor.

### Exercício 3.2:

### 3. Valor

Busque outras definições de valor, principalmente de outras áreas que não foram citadas neste capítulo.

#### **Exercício 3.3:**

Vamos supor que Alice queira abrir um negócio. Para isso ela precisa de um software que permita registrar suas vendas. Sem esse software, por motivos legais, ela não pode vender. No mercado ela pode encontrar um software básico, que não atende todas as suas necessidades mas atende o requisito legal, por R\$ 100.000,00 por ano. Nesse caso, ela perderia a capacidade de mudar seus preços dinamicamente, o que faria com que ela deixasse de ganhar R\$ 240.000,00 por ano. Porém, a taxa de juros do mercado, se ela investir o dinheiro que usaria para o software novo é de 5% ao ano. Quanto ela pode pagar de aluguel, por ano, pelo software que atende completamente suas necessidades?

**Solução:** A princípio ela pode pagar um valor que 5% seja menor que R\$240.000, isto é R\$4.800.000,00. Se ela investir menos que isso vai ganhar mais no total, se investir mais que isso não vai ganhar tanto quanto ganharia se colocasse o dinheiro para render.

#### **Exercício 3.4:**

Vá para o site <http://jogodeanalisedesistemas.xexeo.net/> e visite a Livraria Resolve. Identifique qual pode ser o conceito de valor para o projeto que essa organização deseja.

#### **Exercício 3.5:**

Este trabalho foi inspirado no “Wake up in the morning game”<sup>7</sup>, adaptado para alunos de faculdade e um curso remoto.

- Listar atividades individualmente. (5 minutos)
  - Cada membro do grupo deve usar stick-notes para listar suas atividades de manhã, desde quando acordam até quando chegam à faculdade.
- Com o grupo, agrupar as atividades individuais em tópicos comuns. (5 minutos)
  - Usar stick-notes de outra cor ou tamanho.
  - Agrupar em função do objetivo.
- Ordenar as atividades no tempo, cronologicamente, da esquerda para direita. (5 minutos)
- Priorizar as os grupos de atividade segundo a técnica MoSCoW - de cima para baixo. (5 minutos)
- O que você faria se acordasse atrasado/tarde para uma reunião muito importante? (5 minutos)
  - Cada membro do grupo colocar estrelas nos stick-notes que não podem deixar de fazer.
  - Não tem limite de estrelas.

---

<sup>7</sup><https://www.agilesparks.com/blog/wake-up-in-the-morning-game/>

- Anotem onde o item estrelado muda a prioridade. (5 minutos)
  - Por exemplo, pode ser necessário pegar um táxi, o que antes seria considerado um “*Won’t*”.
  - Pode não existir no seu caso.



# 4

## 5W2H

I keep six honest serving-men  
(They taught me all I knew);  
Their names are What and Why  
and When  
And How and Where and Who.

(Rudyard Kipling)

## Conteúdo

---

4.1.	As Perguntas Certas . . . . .	60
4.2.	Usos do 5W2H . . . . .	61
4.3.	Onde Está o Valor? . . . . .	63
4.4.	Várias Perguntas para cada Palavra . . . . .	64
4.5.	Exercícios . . . . .	67

Este capítulo apresente uma técnica informal, porém extremamente útil, de descrição de fatos, coleta de informações, análise crítica análise de situação, ou registro de decisão conhecida como **5W2H(nakagawa:2012:5w2h)**.

A sigla 5W2H representa as iniciais de sete perguntas direcionadoras: "What"(O que será feito?), "Why"(Por que será feito?), "Where"(Onde será feito?), "When"(Quando será feito?), "Who"(Por quem será feito?), "How"(Como será feito?) e "How much"(Quanto custará fazer?).

Este método foi historicamente usado em várias disciplinas, incluindo engenharia de software, gestão de projetos e outras áreas que requerem planejamento estratégico e

#### 4. 5W2H

operacional. De tão simples, não existe uma referência formal, ou um método específico de como a usar, na área técnica, apesar de vários textos ou artigos a citarem ou proporem formas de usá-la e adaptá-la. Apesar de simples e informal, ela também é usada como base para vários modelos formais, de diferentes graus de complexidade, tanto de forma explícita como implícita.

### 4.1. As Perguntas Certas

Todo trabalho que vai ser feito precisa ser definido de alguma forma. Para isso é importante que se façam as perguntas certas, e que essas perguntas levem a uma definição precisa, ou seja, não ambígua, do que vai ser feito e quais as condições em que será aceito. Também, quando se descreve algo, é necessário que as informações dadas sobre o que está sendo descrito sejam suficientemente completas para serem úteis. Isso é verdade para a criação de notícias, histórias, acordos de ações a serem tomadas no final das reuniões e para a análise de sistemas, onde estamos na prática descrevendo um sistema de forma a permitir sua implementação.

Um quantidade grande de textos, desde daqueles que tentam ensinar crianças a contar histórias até aqueles que ajudam a profissionais a determinar uma arquitetura de informação empresarial, indica que é necessário responder as seguintes perguntas, conhecidas como 5W2H em inglês:

- Por que? (*Why*)
- O que? (*What*)
- Quem? (*Who*)
- Quando? (*When*)
- Onde? (*Where*)
- Como? (*How*)
- Por quanto? (*How Much*)

Existem variações. Uma lista mais básica só possui 6, exclui o “Por quanto?” (5WH). Listas maiores incluem termos como “ganhos” ou *Wins*, “Quantos”, e variações de “Quem” que são usadas em inglês, como “Whose”, em português “De quem?”, e “Whom”, que é uma versão passiva. Na prática, essa lista de 7 é a mais conhecida na área de negócios, onde os valores financeiros são sempre relevantes, e a versão 5WH, em áreas como jornalismo e educação.

A fonte original destas perguntas é o estudo da Ética, e mais tarde da Retórica, na Grécia antiga, e foram chamadas originalmente de **sete circunstâncias** por Aristóteles. Em latim as perguntas eram: *quis, quid, quando, ubi, cur, quem ad modum e quibus adminiculis*. A tradução mais próxima seria: quem, o que, quando, onde, por que, como e por que meios (Fedotov, 2019; Sloan, 2010).

Por que essas perguntas são boas? Primeiro, quando usadas realmente como perguntas em uma entrevista, são perguntas abertas, que não incluem a resposta dentro delas

e obrigam quem responde a responder sem usar apenas uma resposta do tipo “sim” ou “não”. Também são perguntas que, a princípio, não induzem uma resposta. Por exemplo, perguntamos “Quem merece ganhar o prêmio?” em vez de “Alice merece ganhar o prêmio?”, deixando para o entrevistado decidir sem ser influenciado pelo nome de Alice na pergunta. Além disso, as perguntas, reunidas, mostram um grande quadro informativo sobre um tema.

## 4.2. Usos do 5W2H

Um das áreas de utilização, por exemplo, é o jornalismo. Se uma notícia vai ser dada, é importante saber responder, no texto da notícia, todas essas perguntas, ou a notícia estará incompleta.

Um exemplo é o seguinte primeiro parágrafo de uma notícia:

BRASÍLIA — Um dia depois de vetar a proibição para que as companhias aéreas cobrem pelo despacho de bagagens, o presidente Jair Bolsonaro justificou a decisão, afirmado, nesta terça-feira, que “empresas menores alegavam que seria um empecilho” à operação no país e disse que não pretende enviar outra medida ao Congresso Nacional para restringir a cobrança apenas para as chamadas “low cost”, de baixo custo. (Extra, 2019)

As perguntas 5W2H para essa notícia estão respondidas na Tabela 4.1.

Pergunta	Resposta
Onde	Brasília
Quando	Um dia depois de vetar nesta terça-feira
Quem	Jair Bolsonaro
O que	vetar a proibição para que as companhias aéreas cobrem pelo despacho de bagagens
Por que	empresas menores alegavam que seria um empecilho à operação no país

Tabela 4.1.: Perguntas e respostas a partir da notícia.

Outro uso é para registrar decisões de uma reunião. Nesse caso se constrói, em uma folha de papel ou quadro branco, um quadro onde os itens de decisão são descritos e, para cada um deve ser registrado:

- O que deve ser feito?
- Quem é o responsável?
- Quando deve ser feito/entregue?
- Onde deve ser feito/entregue?
- Por que deve ser feito?
- Como vai ser feito?

#### 4. 5W2H

- Quanto vai ser gasto?

Possivelmente, em alguns usos, podem não fazer sentido responder algumas perguntas. Por exemplo, pode não ser adequado responder onde algo vai ser feito se sempre será feito no escritório, ou como será feito se é algo óbvio.

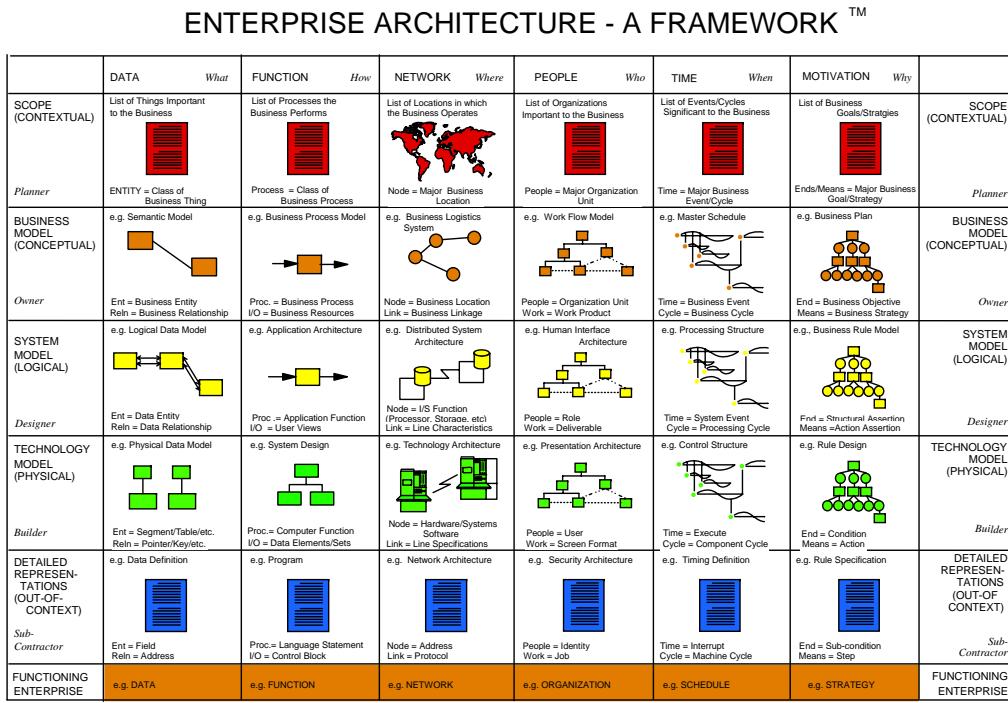
Também é importante notar que a pergunta “Por que” vai ser feito, mesmo que pareça desnecessária, é um guia importante para a forma como algo vai ser realizado. Isto é, para um mesmo “o quê”, vários “como” diferentes podem ser possíveis tendo em visto o motivo da entrega. Ainda mais importante, para alguns autores, é que as pessoas saibam por que estão fazendo suas tarefas (Flores, 2012).

Na análise de sistemas, e na análise de requisitos, as perguntas mais respondidas são ligadas a **quem** deseja **o que** e **por que** deseja. Perguntas adicionais, feitas em algumas práticas ou que tem que ser respondidas nas fases iniciais de um projeto de software são, por exemplo: para **quando** deseja, **por quanto** deseja, **como** deseja que seja feito e **onde** deseja que funcione. Várias outras perguntas podem ser feitas e que também vão influenciar o projeto, como **quem** paga, **quem** vai ser afetado, até **quando** vai funcionar, etc.

##### 4.2.1. O *Framework de Zachman*

O *Framework de Zachman* é uma metodologia de descrição de uma arquitetura de sistemas de informação baseado na técnica 5WH, i.e., não incluindo custos (Sowa e Zachman, 1992; Zachman, 1987). Várias das técnicas que usaremos nesse livro geram artefatos que pertencem ao *Framework de Zachman*. Ele é um dos principais exemplos do uso direto da técnica 5WH em Engenharia de Software.

Sua principal caracterização é o quadro da Figura 4.1.



© 1986 - 2005 John A. Zachman, Zachman International

Figura 4.1.: O Framework de Zachman fornece uma visão da arquitetura empresarial a partir de 6 dimensões definidas pelo 5WH. Reprinted with permission by John A. Zachman, Zachman International.

## 4.3. Onde Está o Valor?

Tendo em vista que a proposta é **desenvolver software tendo em vista entregar valor ao cliente**, onde está o valor no método 5W2H?

Normalmente a resposta é razoavelmente simples: o valor está normalmente caracterizado, de alguma forma na resposta a pergunta “por que?” feita a um requisito, pois a justificativa de um requisito indica o motivo pelo qual ele é importante.

É possível que, para descobrir o verdadeiro valor, seja necessário fazer essa pergunta mais de uma vez, isto é, perguntar por que a resposta para a pergunta “por que” foi dada, e então perguntar de novo até que se alcance uma resposta que realmente caracterize o valor. Essa técnica é conhecida como **5 Por Ques** (Serrat, 2017).

Algumas vezes também é possível que a resposta necessite de perguntas do tipo *O que*, ampliando o conceito tanto do “por que” quanto do “o que”. Por exemplo, se em algum momento a resposta for “preciso dessa função por causa de uma lei”, a próxima pergunta poderia ser “O que (ou qual) é essa lei?”.

## 4. 5W2H

Perguntar por que sucessivamente ou perguntar o que sucessivamente, ou misturar as duas sequências, é uma prática chamada *Squeeze and Stretch Higgins* (1994).

### 4.4. Várias Perguntas para cada Palavra

Algumas perguntas podem ser feitas mais de uma vez. Por exemplo, “Quem?” podem se referir ao agente ou ao paciente de uma ação, ou a um observador. A pergunta “Quando?” pode se referir ao quando começou ou quando acabou o evento, ou quando ele foi planejado.

Cabe ao profissional entender como usar essas possibilidades ao seu favor. As subseções a seguir mostram algumas opções que podem, e na maioria das vezes devem, ser investigadas em um projeto, sem esgotar as possibilidades.

Além disso, como as práticas de desenvolvimento de software tratada nesse livro podem ser usadas desde o início do projeto, para servir de especificação do mesmo, várias perguntas podem ser feitas não só sobre o produto, mas também sobre o projeto.

#### 4.4.1. Perguntas sobre o que

Descobrir o que é a tarefa principal da análise. Basicamente todo este livro fala sobre isso.

Normalmente as perguntas do tipo “o quê” caracterizam mais o produto da análise que o projeto propriamente dito.

- O que será feito?
- O que não será feito?
- O que é necessário?
- O que é suficiente?
- O que é insuficiente?

#### 4.4.2. Perguntas sobre quem

As perguntas sobre “quem” são muito importantes em várias fases do projeto. No início porque precisamos identificar as partes interessadas (Capítulo 5). Depois porque cada funcionalidade do sistema vai atender a algum usuário.

Uma das técnicas muito usadas em gerência de projetos é, para cada atividade, classificar dos as partes interessadas de acordo com suas responsabilidades, criando uma matriz conhecida como Matriz RACI (PMI, 2017). Essa matriz contém pacotes de trabalho ou atividades nas linhas e partes interessadas nas colunas, sendo que cada célula deve indicar se a parte interessada **realiza**, **aprova**, **é consultada** ou **é informada**.

- Quem deve realizar?
- Quem deve aprovar?
- Quem deve ser consultado?
- Quem deve ser informado?
- Quem deve estar envolvido?
- Quem pode ser afetado?
- Quem pode se beneficiar?
- Quem pode ser prejudicado?
- Quem deve verificar?
- Quem deve validar?
- Quem deve homologar?
- Quem deve pagar?
- Quem deve usar?
- Quem deve operar?
- Quem deve manter?

#### 4.4.3. Perguntas sobre quando

Geralmente as perguntas do tipo “quando” tem relação com prazos. Elas podem se refenciar tanto ao negócio quanto ao projeto, ou seja, existem tempos do negócio, como o prazo para entregar um relatório, e tempos do projeto, como o prazo para entregar uma função no software.

- Quando deve ficar pronto?
- Quando deve iniciar a execução?
- Quando vai acontecer?
- Quando será tarde demais?
- Quando os recursos estarão disponíveis?
- Quando devem ser as entregas?
- Quando deve ser aprovado?

#### 4.4.4. Perguntas sobre onde

Novamente esse tipo de pergunta tem relação com diferentes assuntos. Um é o local do mundo real, por exemplo, um software pode ser usado em uma fábrica, mas não no escritório de uma organização. Outro é em relação a infraestrutura de TI, o software pode rodar na nuvem, em um servidor da empresa ou em celulares. Finalmente, em relação ao projeto, um software pode ser feito em um laboratório ou por uma equipe espalhada pelo mundo.

- Onde será feito?
- Onde será testado?
- Onde será homologado?

#### 4. 5W2H

- Onde será instalado?
- Onde será utilizado?
- Onde será mantido?

#### 4.4.5. Perguntas sobre como

Geralmente a fase de Análise, objetivo deste livro, não discute muito o “como”, sendo mais preocupada com o “o quê”, sendo o “como” mais tratado na fase de projeto. Mesmo assim, várias perguntas desse tipo já tem que ser respondidas pela análise, tanto em relação ao produto como ao projeto de construí-lo.

- Como será feito?
- Como será utilizado?
- Como será comercializado?
- Como será corrigido no futuro?
- Como sua operação será gerenciada?

#### 4.4.6. Perguntas sobre por que

Apesar de parecer que os porquês não influenciam a execução do projeto, a verdade é que considerá-los traz uma mudança importante na visão que o analista vai ter da solução. Os porquês são na verdade a justificativa, definem valor e são a orientação do caminho a ser seguido a cada decisão.

- Por que aconteceu?
- Por que será feito?
- Por que foi pedido?
- Por que foi aprovado?
- Por que será aprovado?
- Por que existem resistências?
- Por que alguém é a favor?
- Por que alguém é contra?
- Por que a funcionalidade é desejada?
- Por que o requisito foi pedido?

#### 4.4.7. Perguntas sobre quanto

Não existe projeto sem orçamento. Mesmo que você vá trabalhar para si, ou de graça, há um custo em horas e outros recursos necessários. Por isso essas perguntas são normalmente respondidas quando todas as outras já foram respondidas, ou, ao contrário, são respondidas antes porque vão servir de limites ao projeto, limitando as respostas aceitáveis para as outras perguntas.

- Quanto custará fazer?
- Quanto custaria não fazer?
- Quanto será obtido em benefício quando feito?
- Quanto será necessário do recurso X<sup>1</sup>?
- Quanto tempo será necessário?

## 4.5. Exercícios

### Exercício 4.1:

Vá para o site <http://jogodeanalisedesistemas.xexeo.net/> e visite a Livraria Resolve. A partir da visita, analise as respostas dadas às perguntas 5W2H e veja se elas são adequadas. Escreva uma lista de perguntas adicionais, sempre no formato 5W2H a serem feitas para cada personagem.

---

<sup>1</sup>pessoas, funções no projeto, equipamentos, etc.



# 5

## Partes Interessadas

Find the appropriate balance of competing claims by various groups of stakeholders. All claims deserve consideration but some claims are more important than others.

*(Warren Bennis)*

### Conteúdo

---

5.1.	O Que São Partes Interessadas . . . . .	71
5.2.	Partes Interessadas e Valor . . . . .	75
5.3.	Gerência das Partes Interessadas . . . . .	75
5.4.	A abordagem SEMAT para Partes Interessadas . . . . .	76
5.5.	Caracterização da Parte Interessada . . . . .	76
5.6.	Classificação das Partes Interessadas . . . . .	79
5.7.	Representação das Partes Interessadas . . . . .	83
5.8.	Engajamento das Partes Interessadas . . . . .	83
5.9.	Matriz RACI . . . . .	83
5.10.	Conclusão . . . . .	85

## Por que partes interessadas?

Projetos de software afetam e são afetados por pessoas e organizações, as partes interessadas. Entender como modelar e trabalhar com elas é essencial para conclusão satisfatório do projeto.

Todo projeto, inclusive projetos de software, não só atende a pessoas e organizações, compradores, clientes e usuários, mas também **afetam e são afetados** por outras pessoas e organizações de alguma forma. O somatório de todas essas pessoas, direta ou indiretamente ligadas ao projeto, é chamado de **partes interessadas**<sup>1</sup>.

As partes interessadas são um Alpha do SEMAT, visto na Figura 5.1, no nível de Cliente (Jacobson, Ng et al., 2013). Neste modelo elas fornecem as oportunidades para um projeto de software, demandam requisitos, suportam o time de desenvolvimento e usam e fruem do sistema de software. Na verdade, elas tem a posição central de todo projeto. O time, na prática, trabalha para as partes interessadas.

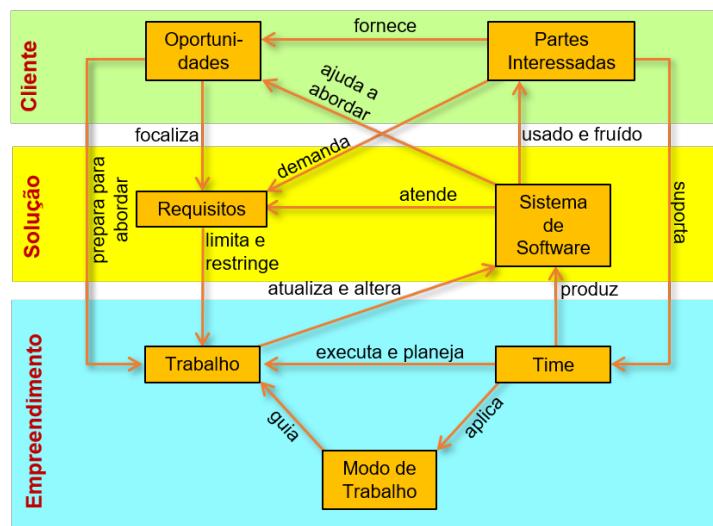


Figura 5.1.: O Kernel Essencial do SEMAT. Fonte: (Jacobson, Ng et al., 2013)

A influência das partes interessadas em um projeto tem um comportamento inverso ao do custo de mudanças, como ilustra a Figura 5.2. Ao longo do projeto, como o trabalho vai ficando pronto, uma mudança provocada pelas partes interessadas começa a custar muito mais, logo elas vão perdendo a capacidade de fazer alterações. Por outro lado, no início, quando o projeto ainda não fez escolhas e compromissos importantes que determinarão o seu futuro, e as mudanças custam pouco, as partes interessadas têm grande influência.

<sup>1</sup>O nome em inglês, **stakeholders**, aqueles que possuem algum interesse ou aposta no projeto, é também usado.

A tudo isso se soma o fato que um objetivo geral de qualquer projeto é **satisfazer as partes interessadas**.

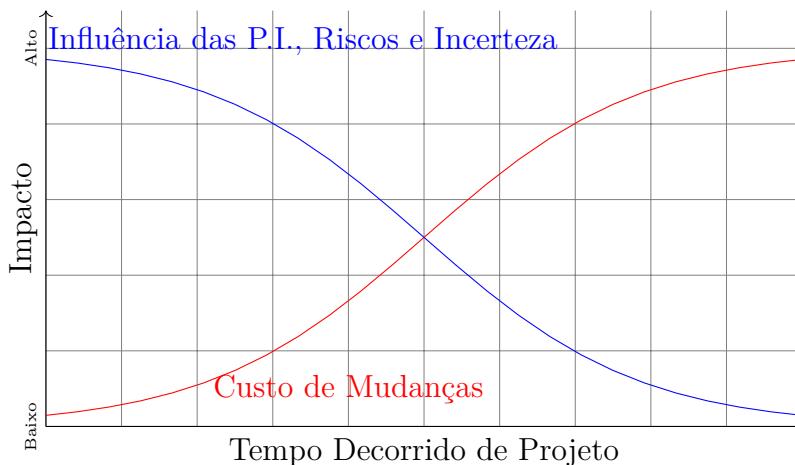


Figura 5.2.: O impacto das partes interessadas.

As práticas ágeis, porém, tentam diminuir, ao mesmo tempo, tanto a perda de influência das partes interessadas, quanto o aumento do custo de modificações. Isso é feito principalmente com os usuários e clientes mais próximos do projeto, ao longo do tempo. Mesmo assim, grandes mudanças têm um custo maior e podem ficar inviáveis.

## 5.1. O Que São Partes Interessadas

Uma boa definição do que é uma parte interessada é fornecida pelo Guia PMBOK: “é um indivíduo, grupo ou organização que pode afetar, ser afetada ou sentir-se afetada por uma decisão, atividade ou resultado de um projeto. As partes interessadas do projeto podem ser internas ou externas ao projeto, e podem estar envolvidas ativamente ou passivamente, ou mesmo não estar cientes do projeto” (PMI, 2017). Também é possível, para facilitar uma análise, falar de outros tipos de parte interessada, não humanos, como um ecossistema ou o meio-ambiente<sup>2</sup>.

O PMBOK (PMI, 2017) exemplifica várias partes interessadas em um projeto:

- equipe do projeto:
  - gerente do projeto;
  - equipe de gerência do projeto;
  - outros membros do projeto;
- patrocinador;
- clientes e usuários;
- fornecedores;

<sup>2</sup>No Project Model Canvas, Capítulo ??, há o conceito de Fatores Externos, aplicável nestes casos.

## 5. Partes Interessadas

- parceiros comerciais;
- gerentes funcionais;
- gerente de portfólios;
- gerente de programas;
- escritório de projetos, e
- outras partes interessadas.

Esse mesmo guia avisa que “As partes interessadas do projeto podem ter um impacto positivo ou negativo no projeto, ou ser impactadas de forma positiva ou negativa pelo projeto.” (PMI, 2017).

O grupo dos que afetam e são afetados, com conhecimento de causa ou não, é bem maior que os envolvidos diretamente no projeto. Por exemplo, em um sistema que controla um radar, dedicado a aplicar multas de trânsito automaticamente, podem ser afetados os guardas de trânsito, que podem perder o emprego, os passageiros de todos os veículos e até todo o sistema de saúde do país, devido a diminuição da gravidade dos acidentes. Também podem ser afetados os moradores da região, com menos atropelamentos, ou com um trânsito mais lento que atrasa seus horários. Outro exemplo pode ser um sistema de marcação e monitoração do momento de atendimento de consultas médicas, onde os acompanhantes dos pacientes também são afetados.

Uma avaliação da Tabela 2.2 mostra a importância das partes interessadas para o sucesso de um projeto. Entre os 10 primeiros fatores de sucesso, 5 são ligados a pessoas com interesse no projeto. Entre os 4 primeiros, que estão em igualdade de condição, o envolvimento do Patrocínio Executivo e dos Usuários se mostra essencial.

### Nunca diga “usuário”

Gladys S. W. Lam (2015) diz que um dos princípios básicos da análise de negócio é **nunca dizer usuário**. Segundo ele, usuários só existem na perspectiva de um sistema de TI. Nos contextos de negócio as pessoas não são usuárias, mas atores centrais do negócio. Na prática, usuário não significa muito em um mundo onde todos são usuários de sistemas de TI, então ao tratar as partes interessadas devemos usar o termo específico do negócio, como paciente, atendente, médico e enfermeira em um ambulatório, e correntistas, caixa e gerente em um banco.

Deve ficar claro que todo projeto se inicia por meio de das partes interessadas, porém não necessariamente de todas. Descobrir todas as partes interessadas é uma das missões iniciais de um projeto. Na maioria das vezes os usuários, pelo menos em sua maioria, estão identificados no inicio do projeto. Outro caso comum é uma parte interessada advocar a necessidade do projeto para trazer melhorias para outra parte. É comum que software seja desenvolvido para melhorar o atendimento ao cliente da organização que está adquirindo o produto, obviamente a espera de melhorar o resultado global.

Entre as partes interessadas, o usuários do software sendo produzido são um grupo importante, que merece destaque especial em nossos estudos. A princípio são eles,

possivelmente por meio de uma representação, que nos fornecerão os requisitos e aceitarão o produto.

### 5.1.1. Usuários

Desenvolvemos sistemas para serem usados. Na verdade, uma medida importante da qualidade de um sistema de informação e do sucesso do projeto que o construiu é quanto do sistema é realmente usado pelos seus usuários, ou seja, quanto do que foi feito é realmente útil. Logo, para descobrir o que deve ser feito (o quê), devemos primeiro entender para quem o sistema se destina (quem) e ainda entender o motivo dessas pessoas(por que).

Um tipo especial de parte interessada é o **usuário**. Como diz o nome, um usuário sempre **usa** o sistema. Esse nome pode ser entendido em duas formas. Na forma restrita, indica o **usuário final**, isto é, os que realmente usam o sistema dentro do escopo do seu objetivo, interagindo com suas interfaces, sejam por meio de telas ou relatórios impressos. Já na definição de forma ampla, todos aqueles que usam alguma parte do sistema, incluindo sua documentação e suas estruturas de dados, como os desenvolvedores, que usam o código do sistema, ou os responsáveis por sua operação, também são usuários.

Mesmo com uma interpretação ampla, algumas partes interessadas não são usuários, pois nunca usam alguma parte do sistema, mesmo que sejam afetadas por eles.

Novamente, segundo o PMBOK, o objetivo de todo projeto é gerenciar a satisfação do usuário (PMI, 2017). Por isso é importante garantir o envolvimento de todas as partes interessadas. A forma de fazer isso é por meio de um processo de comunicação contínua, para “entender necessidades e expectativas, abordar as questões conforme elas ocorrem, administrar os interesses conflitantes e incentivar o engajamento apropriado das partes interessadas com as decisões e atividades do projeto” (PMI, 2017).

### 5.1.2. Perspectivas dos Usuários

As perspectivas básicas que encontramos em entrevistas e reuniões com usuários, são três, como ilustrado na Figura 5.3:

1. o usuário onisciente;
2. o usuário externo ao sistema,e
3. o usuário interno ao sistema.

O **usuário onisciente** fala do sistema como um todo, indicando normalmente coisas que ele deve, ou deverá, fazer. Ele vê tanto o sistema quanto seus usuários de uma perspectiva externa, conhecendo os mecanismos, de forma abstrata, tanto por dentro quanto por fora. Geralmente ele possui uma posição de gerência e muitas vezes é o patrocinador executivo do projeto. Algumas vezes já

## 5. Partes Interessadas

É possível que ele não saiba como o sistema funciona realmente, pois seu contato pode ser de outra época em relação a entrevista, desconhecendo então modificações e detalhes atuais. Ele normalmente exige funcionalidade gerencial do sistema. Esse usuário muitas vezes quer, e é capaz de ajudar a definir, uma nova abordagem de negócio para o sistema. Tipicamente é um gerente ou diretor e tem algum poder sobre os usuários com outras perspectivas.

O **usuário externo** descreve o sistema como se o estivesse usando, e normalmente já usa uma versão do sistema. Exige funções do sistema, mas principalmente para atender seu nível de atuação, seja gerencial ou operacional. Ele pode apresentar algum nível de desconfiança, pois o novo sistema pode exigir um esforço para conseguir novos conhecimentos ou distribuir conhecimento que hoje apenas ele tem acesso, o que afeta o poder dele na organização. Além disso, ele conhece bem a entrada e a saída do sistema atual, mas não necessariamente os procedimentos e algoritmos, ou seja, como as contas são feitas. Esse usuário, principalmente quando mais próximo do nível operacional da organização, pode levar ao desenvolvimento de um sistema igual ao já existente, apenas com a correção dos defeitos, sem mudança na prática do negócio.

O **usuário interno** descreve o sistema visto por dentro. Algumas vezes, não existe ainda um sistema plenamente informatizado, e é esse usuário que faz a parte de cálculos e tomadas de decisão. Pode apresentar um risco ao desenvolvimento, porque ele é candidato a substituição pelo sistema novo, o que pode gerar desconfiança, franca hostilidade e mesmo sabotagem do projeto. Ele conhece normalmente muito bem como o sistema atual funciona por dentro.

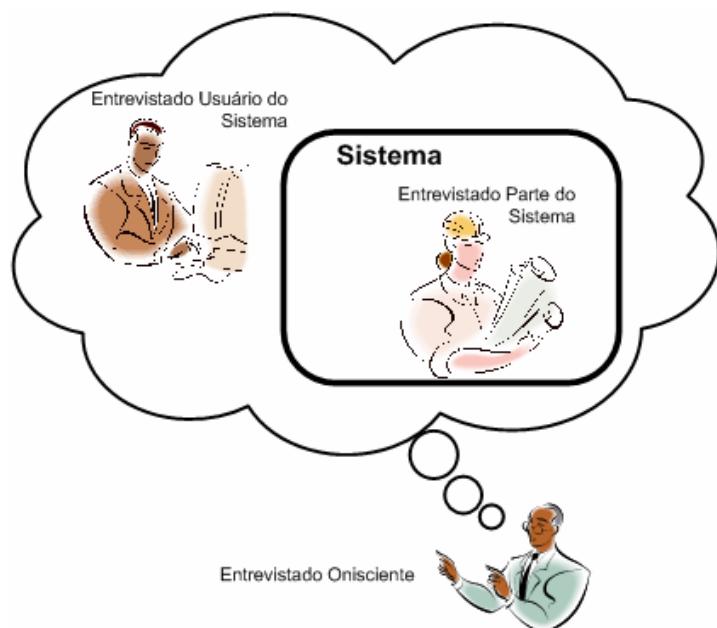


Figura 5.3.: 3 tipos comuns de usuários dos sistemas. Fonte:próprio autor

## 5.2. Partes Interessadas e Valor

Todo valor do projeto é entregue às partes interessadas. Na verdade, seria mais correto dizer que “todos os valores do projeto” são entregues às partes interessadas, já que um projeto pode ter valores diferentes de acordo com as partes interessadas, pois elas possuem interesses diferentes, como será tratado na Seção 5.5.1.

Suponha o seguinte projeto de software: uma conjunto de serviços B2B para uma editora, que serão usados por livrarias *on line* que desejam comprar produtos *on demand* quando forem vendidos em suas lojas virtuais B2C. Esse software está planejado para ter efeito de melhorar o serviço das grandes livrarias *on line*, prejudicando, por consequência, as pequenas livrarias locais. Então, podemos concluir que ele fornece valor positivo para algumas partes interessadas:

- a editora;
- grandes livrarias *on line*;
- o leitor;
- o escritor;
- o serviço de entrega, e
- outros atores, como pessoal de operação do software, vendedores de equipamentos, etc.

Porém ele também fornece um valor negativo para partes interessadas que são afetadas, principalmente de forma indireta, como:

- livreiros e trabalhadores de livraria, que podem perder o emprego;
- proprietários que alugam lojas para livrarias;
- pessoas que querem comprar um presente pessoalmente em cima da hora, e
- pequenas livrarias locais.

É claro que no mercado esse padrão sempre se repetirá: favorecer um negócio sempre prejudica seus competidores, e com isso toda a sua cadeia de produção. Normalmente não há um problema legal ou ético com isso, mas é possível que em condições limites alguma lei de proteção à livre concorrência possa ser ferida.

## 5.3. Gerência das Partes Interessadas

A gerência das partes interessadas aparece em vários documentos da área de Gerência de Projeto e de Engenharia de Software.

Ela é uma atividade do PMBOK que inclui (PMI, 2017):

- “processos para identificar pessoas, grupos ou organizações que podem impactar ou ser impactadas pelo projeto”;
- “análise das expectativas da partes interessadas e seu impacto no projeto”, e

## 5. Partes Interessadas

- “o desenvolvimento de estratégias apropriadas de gerência para efetivamente engajar as partes interessadas nas decisões e execução do projeto”.

Para isso é necessária a comunicação contínua com todas as partes interessadas, no intuito de entender e registrar todas as necessidades e expectativas. Além disso, é necessário abordar todas as questões e conflitos de interesse, que podem acontecer a qualquer momento do projeto, e ainda promover o engajamento apropriado nas decisões e atividades do projeto (PMI, 2017).

Essa gestão, então, vai se resumir em 4 partes (PMI, 2017):

1. identificar as partes interessadas;
2. planejar a gerência das partes interessadas;
3. gerenciar o engajamento das partes interessadas, e
4. monitorar o engajamento das partes interessadas.

## 5.4. A abordagem SEMAT para Partes Interessadas

O SEMAT já considera que o trabalho com as partes interessadas deve evoluir em uma escala de 6 estados, do menos ao mais sofisticado (Jacobson, Ng et al., 2013):

1. reconhecer as partes interessadas;
2. criar uma representação para as partes interessadas;
3. envolver as partes interessadas;
4. fazer com que as partes interessadas concordem;
5. satisfazer as partes interessadas para a implantação, e
6. satisfazer as partes interessadas com o uso.

## 5.5. Caracterização da Parte Interessada

Após a identificação das partes interessadas, continua-se o levantamento e caracterização.

Para isso, normalmente são respondidas, para cada parte interessada, algumas perguntas, como:

1. Nome
2. Posição na organização
3. Posição/Responsabilidade no projeto
4. Interesse no projeto (Stake)
5. Impacto no projeto
6. Benefícios recebidos do projeto
7. Colaboração necessária ao projeto (o que precisamos dessa parte interessada)?
8. Atitudes percebidas (riscos)

Essas perguntas variam de um autor da área para outro, porém sempre é necessário ter uma visão de como a parte interessada pode impactar o projeto, sendo apoiando, sendo criando obstáculos. Isso pode ser feito com práticas de classificação das partes interessadas.

### 5.5.1. Interesses e Objetivos

Como diz o próprio termo, todas as partes interessadas possuem interesses no sistema, mesmo que não estejam cientes do mesmo, ou do seu planejamento. Os interesses têm relação com as expectativas, explícitas e implícitas, que o sistema, em operação, afete a elas, o seu domínio, ou o seu negócio, de alguma forma. Interesses são benefícios que as partes interessadas esperam obter do sistema.

Os usuários finais possuem também **objetivos**, i.e., eles usam o software para obter uma, ou mais, resposta planejada. Neste contexto, objetivos são requisitos funcionais que os usuário esperam encontrar no sistema.

A princípio, se deseja que o sistema seja construído para atingir seus objetivos de forma que todos os interesses sejam atendidos. Porém, pode haver incompatibilidades tanto entre os interesses quanto entre os objetivos das diversas partes interessadas. Tratar casos como esse já foi discutido inclusive como um processo de software específico, conhecido como WinWin (B. W. Boehm, 1988; B. Boehm e Kitapci, 2006).

É importante atentar para a diferença que existe entre **o uso que uma parte interessada fará do sistema** (o que), o seu **objetivo**, com o **interesse** que uma parte interessada tem no sistema, que é o **resultado esperado, ou expectativa** que ela tem. Enquanto todas as partes interessadas tem interesses, só algumas tem objetivos, e essas são as usuárias do sistema.

Estudar os interesses nos permite saber melhor como realizar os objetivos. Por exemplo, um cliente de web site de vendas de produtos tem como objetivos, ao longo de uma compra, várias ações, como escolher um produto, saber seu preço, pagar o produto. Porém ao longo de todo esse tempo, seu único interesse é receber o produto. Mais do que comprar, o usuário quer receber-lo em casa. Então toda a dinâmica do site, ou do software e logística do sistema por trás dele, pode ser construída voltada para esse interesse. Mesmo que quiséssemos atender um interesse menor, como “comprar o produto”, já poderíamos imaginar formas diferentes de implementar as ações correspondentes. Um exemplo disso no mundo real é o *one-click buy*, ou seja, comprar um produto com só um click do mouse e ter certeza de receber-lo, ideia que foi inclusive patenteada pela Amazon<sup>3</sup>.

Como documentação dos interesses e objetivos das partes interessadas, dois formatos são possíveis. O primeiro é uma tabela com três colunas: parte interessada, interesses e objetivos. Essa tabela tem como defeito não identificar claramente partes interessadas

---

<sup>3</sup>O que causou muita polêmica na época. Essa patente já expirou.

## 5. Partes Interessadas

com interesses ou objetivos comuns. A vantagem é a facilidade de gerenciar e preencher, pois é um formulário muito simples.

Tabela 5.1.: Exemplo de formulário para registro do interesse e objetivo da parte interessada.

<b>Parte Interessada</b>	<b>Interesses</b>	<b>Objetivos</b>
Professor	Não perder seus livros Recuperar livros emprestados Saber com quem está um livro Saber os livros que tem	Registrar um empréstimo Registrar uma devolução Registrar um livro Cobrar um livro atrasado
Aluno	Pegar livro emprestado com facilidade Não ser cobrado por livro que devolveu	Registrar um aluno Registrar uma devolução Receber recibo de devolução
Desenvolvedor	Fazer projeto final	

A segunda forma é uma matriz Partes Interessadas vs. Interesses e Objetivos. Dependendo do escopo do sistema ou da quantidade de partes interessadas, escolhe-se um ou outro para coluna ou linha. Nesse caso, a célula que mostra o cruzamento entre parte interessada e interesse ou objetivo pode conter uma avaliação qualquer, como o uso de palavras em uma escala de importância para aquele usuário: baixa, média e alta.

Tabela 5.2.: Exemplo de matriz de cruzamento para registro do interesse e objetivo da parte interessada.

		Professor	Aluno	Desenvolvedor
Interesses	Não perder seus livros	✓		
Objetivos	Registrar um empréstimo	✓		
Recuperar livros emprestados	✓			
Saber com quem está um livro	✓			
Saber os livros que tem	✓			
Pegar livro emprestado com facilidade		✓		
Não ser cobrado por livro que devolveu		✓		
Fazer Projeto Final			✓	
Registrar uma devolução		✓	✓	
Registrar um livro		✓		
Cobrar um livro atrasado		✓		
Registrar um aluno		✓		
Receber recibo de devolução			✓	

## 5.6. Classificação das Partes Interessadas

Além disso, a maioria dos autores propõe classificar as partes interessadas de alguma forma, e contando com apoio de gráficos para entender melhor como gerenciá-las ao longo do projeto.

Um gráfico típico, mostrado na Figura relaciona o interesse da parte interessada com o poder, e indica que tipo de ação deve ser tomada na gerência de partes interessadas.

Essa classificação pode ser feita de várias formas, sendo as mais reconhecidas:

- Poder × Interesse;
- Poder × Influência;
- Influência × Impacto, e
- Poder, Legitimidade e Urgência.

Outra classificação feita é a **Matriz de Engajamento**. Nela se faz uma análise do comportamento atual do usuário em relação a um projeto e o comportamento desejado. São usadas cinco classes possíveis para descrever o engajamento do usuário:

## 5. Partes Interessadas

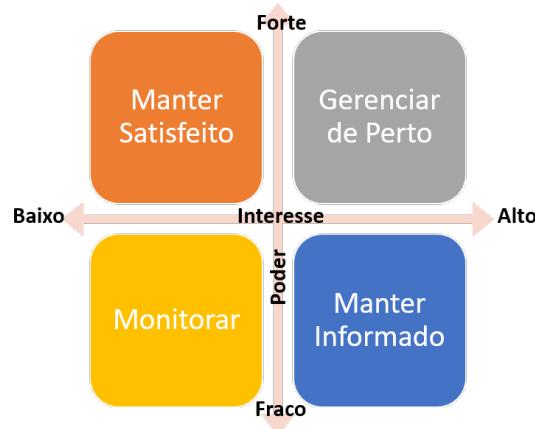


Figura 5.4.: Quadro para o tratamento das partes interessadas. Fonte: (PMI, 2017)

1. desconhece;
2. resistente;
3. neutro;
4. apoiador, e
5. liderança.

O trabalho da equipe é identificar o comportamento atual e, por meio de ações positivas, levar as partes interessadas a um comportamento desejado.

Desconhece	Resistente	Neutro	Apoiador	Líder
Parte Interessada 1	C			D
Parte Interessada 1			C	D
Parte Interessada 1			CD	

Tabela 5.3.: Matriz de Engajamento: C significa comportamento corrente, D o comportamento desejado.

K. E. Wiegers e Beatty (2013), por sua vez, faz uma classificação hierárquica das partes interessadas em:

- clientes, e
  - usuários diretos e indiretos, e
    - ◊ classes de usuários favorecidas, cuja satisfação está fortemente alinhada com os objetivos de negócio do projeto;
    - ◊ classes de usuários desfavorecidas, que não podem usar o produto, por motivos legais, de segurança ou outros, como estelionatários em um sistema bancário;
    - ◊ classes de usuários ignoradas, que não são relevantes ao projeto e
    - ◊ outras classes de usuário, que não se encaixam em nenhuma das outras, mas ainda fornecem requisitos para o projeto .

- outros clientes.
- outras partes interessadas.

### 5.6.1. Análise das Partes Interessadas

A Análise das Partes Interessadas é uma atividade com o objetivo determinar quais são as partes interessadas, e determinar como são afetadas e como podem afetar o projeto, além de entender quem realmente “conta” em um projeto (Mitchell, Agle e Wood, 1997).

Normalmente análise das partes interessadas se inicia com a identificação das mesmas, como mostram PMI (2017) e Jacobson, Lawson e Ng (2019). Muitas são as formas de identificação, mas certamente a experiência do analista será muito importante, além de documentos formais do projeto.

É importante notar que algumas partes interessadas são essenciais ao projeto de software. Por exemplo, deve haver um responsável final pelo financiamento do projeto, normalmente conhecido como patrocinador. Deve também existir pelo menos um grupo de usuários do futuro sistema, os usuários propriamente ditos.

Técnicas de levantamento podem incluir não só a leitura de documentos e a experiência de pessoas da área de negócio, mas também reuniões de *brainstorm* podem ser usadas para levantar o conjunto de partes interessadas a partir de um núcleo inicial de pessoas.

Existe uma base de análise para identificar partes interessadas (Mitchell, Agle e Wood, 1997) e determinar um grau de atenção que deve ser dado a elas, baseada no conceito de **saliência**, que é formada em 3 atributos: Poder, Legitimidade e Urgência.

O **poder** é a “capacidade de um ator fazer com que outro ator realize algo que não realizaria a princípio” (Mitchell, Agle e Wood, 1997). O Poder por ser baseado em coerção, i.e. o uso de força ou ameaça, em utilitarismo, i.e. o uso de incentivos, e normativo, por influências simbólicas.

A **legitimidade** é a “percepção generalizada ou premissa que as ações de uma entidade são desejáveis, apropriadas dentro de algum sistema socialmente construído de normas, valores, crenças e definições” (Mitchell, Agle e Wood, 1997) Ela se baseia no indivíduo, na organização ou na sociedade.

A **urgência** é “grau que a parte interessada reivindica de atenção imediata” (Mitchell, Agle e Wood, 1997) Ela se baseia na sensitividade ao tempo ou na criticalidade, a importância da reivindicação ou do relacionamento com a parte interessada.

Finalmente, a saliência de uma parte interessada “é o grau que o gerente dá prioridade a partes interessadas que competem” (Mitchell, Agle e Wood, 1997). A análise da posse desses três fatores também leva a denominações como as apresentadas na Figura 5.5.

Mitchell, Agle e Wood (1997) afirma que partes interessadas que possuem apenas uma dessas características normalmente são deixadas de lado pelos gerentes, que aumentam o

## 5. Partes Interessadas

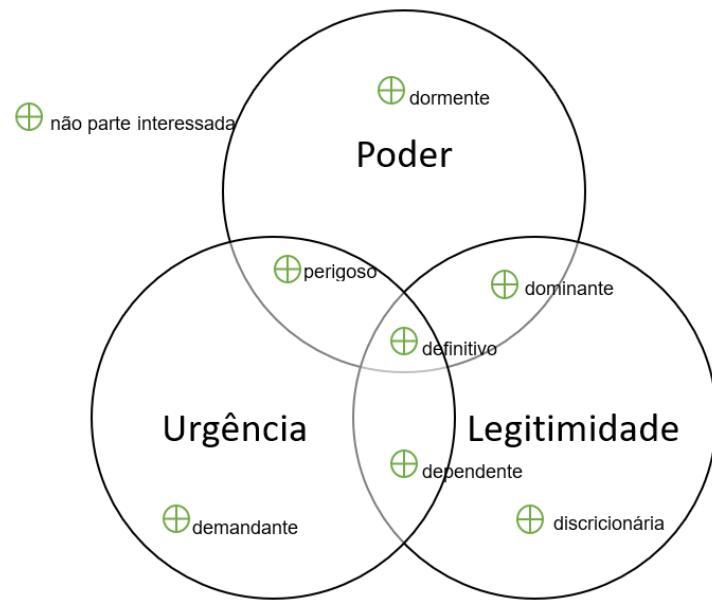


Figura 5.5.: Classificação das partes interessadas. Fonte: (Mitchell, Agle e Wood, 1997)

seu grau de engajamento com as que tem duas das características, até chegar ao grupo dos “definitivos”, que tem prioridade máxima.

## 5.7. Representação das Partes Interessadas

Algumas das partes interessadas são formadas por grupos de pessoas. Dependendo do tamanho do grupo, não é possível trazer todos eles para participar do projeto. Nesse caso é necessário criar uma representação para as partes interessadas (Jacobson, Lawson e Ng, 2019). Caberá a essa representação se envolver e tomar decisões em nome do grupo.

Em projetos ágeis é comum que exista um representante único de todo os usuários, que estabelece as demandas e prioridades do projeto, como o *Product Owner* (Rubin, 2013), porém isso não elimina a existência de todas as outras partes.

## 5.8. Engajamento das Partes Interessadas

O engajamento das partes interessadas se faz por meio de três processos (PMI, 2017):

- inclusão, aumentando o nível de participação das partes interessadas;
- materialidade, pela identificação das questões mais importantes da organização e das partes interessadas, e
- responsividade, pela resposta da organização.

Esse engajamento pode ser obtido de várias formas, dependendo do nível de engajamento que é desejado.

## 5.9. Matriz RACI

A participação das partes interessadas em cada atividade do projeto pode ser planejada por meio de uma Matriz RACI<sup>4</sup>.

A Matriz RACI, também discutida brevemente no Capítulo ??, faz o cruzamento entre as partes interessadas e alguma forma de entender o projeto, como atividades, casos de uso, histórias do usuário, entregáveis, etc. Nesta seção vamos usar o exemplo de entregáveis, sem perda de generalidade.

Para cada um dos entregáveis, deve ser definida, para cada uma das partes interessadas, uma ação, ou mais, em relação ao projeto. Estas ações formam o acrônimo RACI e significam:

- Responsável, que realiza a atividade;
- Aprovador, que aceita a atividade;

---

<sup>4</sup>Apesar de ter uma preocupação em descobrir os autores de cada método proposto neste texto, a matriz RACI parece ser um mistério. Ela pode ser uma adaptação das *Responsability Charts*, propostas por Andersen, Grude e Haug (2009), originalmente na década de 80, ou uma adaptação de uma *Responsibility Assignment Matrix* que aparece citada em documentações da NASA já em 1977 (Facetation, 2015)

## 5. Partes Interessadas

- Consultado, que fornece informações para a realização da atividade, e
- Informado, que deve ser informado sobre a atividade.

A Tabela 5.4 é um exemplo de Matriz RACI.

	Diretor	Gerente	Caixa	Contador	Time Devel
Retirar Dinheiro	I	A	C	I	R
Depositar Dinheiro	I	A	C	I	R
Aprovar Empréstimo	AC	C	I	I	R
Cobrar Dívida	AC	C	I	C	R
Gerar Partidas Dobradas	I	I	I	CA	R

Tabela 5.4.: Exemplo de Matriz RACI

A Matriz RACI tem duas regras importantes:

1. Células podem conter mais de uma letra;
2. Apenas uma pessoa pode aprovar uma atividade, e
3. Pelo menos uma pessoa deve ser responsável por realizar uma atividade.

Além disso, duas análises devem ser feitas para verificar a Matriz RACI.

### 1. A Análise Vertical:

- alguma parte interessada tem um excesso de R's, sendo responsável por uma carga de trabalho excessiva?
- alguma parte interessada tem um excesso de A's, havendo pouca delegação ou responsabilidade excessiva?
- uma parte interessada não tem nenhuma célula em branco, participando de todas as atividades do projeto?

### 2. A Análise Horizontal

- toda atividade tem pelo menos um responsável;
- toda atividade tem um e apenas um aprovador;
- atividades com mais de um responsável devem ser analisadas para duplicidade de trabalho;
- atividades são realizadas e aprovadas em níveis hierárquicos corretos?
- está correto onde o realizador é a mesma pessoa que o aprovador?
- o número de pessoas sendo informada ou consultada é grande ou pequeno demais?

## 5.10. Conclusão

As partes interessadas são importantes por dois motivos principais: necessidade de identificar com quem levantar os requisitos e a necessidade do apoio delas durante o projeto.

Para isso um conjunto de técnicas está disponível. Algumas dessas técnicas, como a Matriz de Engajamento, podem causar problemas se divulgadas fora de um âmbito muito restrito no projeto, outras, como a Tabela RACI fazem parte de documentos ostensivos e precisam ser validadas pelas partes interessadas.

É importante que as partes interessadas sejam gerenciadas ao longo do projeto, seguindo procedimentos tradicionais ou ágeis, e o estado dessa gerência pode ser controlado tanto por técnicas tradicionais de gerência de projeto como por um modelo como o do SEMAT.

## 5.11. Exercícios

### Exercício 5.1:

Descreva as partes interessadas do Sistema de Gestão Acadêmica da sua instituição de ensino.

### Exercício 5.2:

Vá para o site <http://jogodeanalisedesistemas.xexeo.net/> e visite a Livraria Resolve. A partir da sua visita:

1. identifique todas as partes interessadas;
2. classifique-as quanto a:
  - a) poder × interesse;
  - b) poder × influência;
  - c) poder, legitimidade e urgência.
3. construa a Matriz de Engajamento.



# 6

## Qualidade

### Conteúdo

---

6.1.	Tipos de Qualidade . . . . .	90
6.2.	Dimensões da Qualidade . . . . .	90
6.3.	Qualidade é Sempre por um Ponto de Vista . . . . .	95
6.4.	Qualidade de Produto e Qualidade de Processo . . . . .	96
6.5.	Falhas e Defeitos . . . . .	96
6.6.	Quanto Custa a Qualidade . . . . .	97
6.7.	Gestão da Qualidade . . . . .	99
6.8.	Exercícios . . . . .	99

Toda as áreas da Engenharia de Software têm como pedra basilar a Qualidade. Tratada de forma específica, a **Qualidade de Software** é uma das áreas do conhecimento do SWEBOK(IEEE Computer Society, 2014a), onde é dividida conceitualmente de acordo com o mapa mental mostrado na Figura 6.1.

Pressman e Maxim (2014), por exemplo, tentam representar isso com um diagrama piramidal que é de certa forma base para seu livro de Engenharia de Software. Recriado na Figura 6.2, o diagrama mostra como o foco em qualidade suporta os Processos de Software, que por sua vez suportam os Métodos que suportam as Ferramentas.

Como já está claro o que é software, é importante também entender o que quer dizer qualidade. Qualidade é um área de conhecimento que aparece, por necessidade, em muitas outras áreas. Um dos principais autores que discute qualidade, Juran, apresenta duas definições, a primeira sendo “a característica dos produtos que atendem as necessidades

## 6. Qualidade

dos clientes e, assim, proporcionar a satisfação do mesmo” (Defeo e Juran, 2010), e a segunda “a ausência de defeitos” (Defeo e Juran, 2010). A primeira é uma definição focada no uso, a segunda focada no produto em si. Essas duas abordagens são comuns e adotadas na ISO 25010, a norma internacional de qualidade de software (ISO, 2011).

Várias são as perspectivas de entender a palavra qualidade, o que leva os autores muitas vezes optar por mais de uma definição. O SWEBOK, por exemplo, diz que o termo Qualidade de Software pode se referir a “as características desejadas de produtos de software, a extensão em que um produto de software em particular possui essas características e aos processos, ferramentas e técnicas que são usadas para garantir essas características” (IEEE Computer Society, 2014a).

A International Standard Organization (ISO) também tem grande preocupação, em suas normas, com a questão da qualidade. As normas genéricas de qualidade estão na família ISO 9000. Na norma ISO 9000:2015 **qualidade** é definida, de forma bem geral, como “o grau em que um conjunto de características cumpre os requisitos” (ISO, 2015b).

Especificamente sobre Qualidade de Software, existiam as famílias de normas **ISO 9126** e que foram substituídas pelas normas **ISO 25000**, que definem qualidade como “a capacidade do produto de software em satisfazer as necessidades implícitas e explícitas quando usado em condições específicas”(ISO, 2001, 2011).

Segundo o SWEBOK, a área de conhecimento Qualidade de Software trata das questões estáticas de qualidade, enquanto a área Teste de Software trata das questões dinâmicas. A maioria dos autores não vê essa divisão, pelo menos não de maneira tão forte, e a maioria dos congressos sobre Qualidade de Software possui muitos artigos sobre Testes. Na verdade, é possível dizer que os autores do SWEBOK quiseram chamar a atenção da área de Teste de Software colocando ela a parte, mas sem deixar de referenciar que o objetivo dos testes é alcançar qualidade(IEEE Computer Society, 2014a). Neste texto as áreas de Qualidade e Teste serão vistas em momentos distintos.

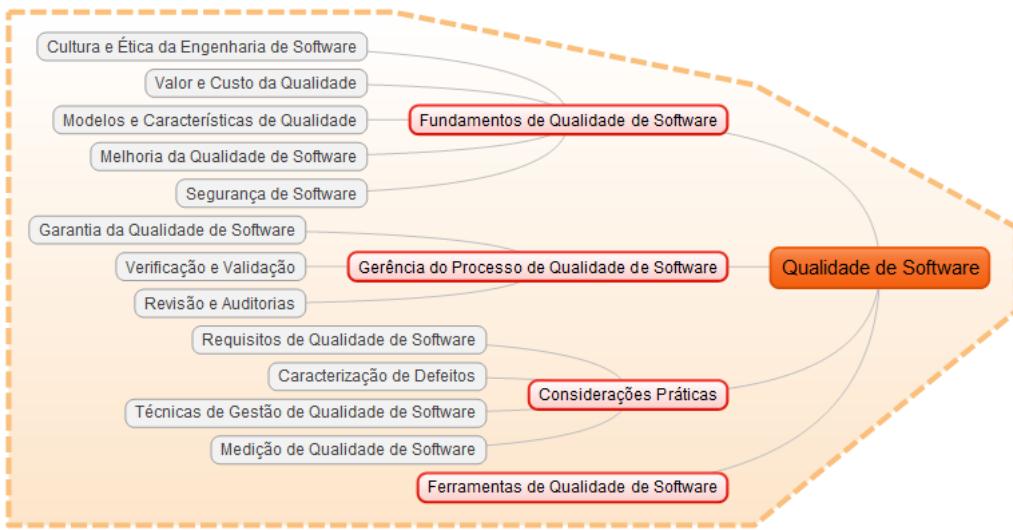


Figura 6.1.: Mapa Mental da organização da área do conhecimento Qualidade de Software no SWEBOK 3.0. Fonte:(IEEE Computer Society, 2014a)

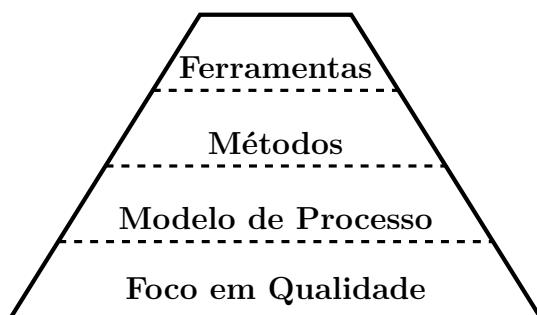


Figura 6.2.: A Qualidade é a pedra basilar da Engenharia de Software. Fonte:(Pressman e Maxim, 2014)

## 6.1. Tipos de Qualidade

Garvin (1984) apresenta 5 significados diferentes que damos a qualidade:

1. **Abordagem transcendental:** onde a qualidade é um atributo global que não pode ser definida precisamente. É a “qualidade sem um nome”, que é reconhecida pela experiência total que uma pessoa tem com o que está sendo avaliado. Seria um conceito primitivo, que não pode ser definido. Um problema dessa abordagem é justamente essa indefinição formal, pois a análise detalhada e objetiva não é possível, sendo uma visão subjetiva e pessoal. Alguns autores usam o acrônimo **QWAN**, para “Quality Without a Name”.
2. **Abordagem baseada no produto:** nesse caso, a definição de qualidade supõe que ela pode ser precisamente definida e medida, a partir de propriedades possuídas pelo produto, pela presença de uma quantidade de certo atributo, como quantidade de gordura no leite. Essa abordagem permite construções da qualidade em diferentes dimensões que devem ser agregadas para um resultado final. Um problema dessa abordagem é da importância de cada dimensão. Nesse caso, é fácil avaliar cada dimensão separadamente, por meio de critérios objetivos, mas é mais difícil de agregar todas essas avaliações. Além disso, supõe que a qualidade não está ligada a experiência, uso ou usufruto do produto, mas sim diretamente a característica do mesmo.
3. **Abordagem baseada no usuário:** onde o que importa é o atendimento às necessidades ou desejos de diferentes usuários. Um problema dessa abordagem é a dificuldade de agregar a visão de vários usuários. Desse modo, mesmo que o objeto não cumpra exatamente a especificação, ainda pode atender o usuário.
4. **Abordagem baseada na fabricação:** onde o que importa é a conformidade com os requisitos definidos para o produto, ou seja, a ausência de desvio em relação a uma especificação. Muito aplicável a processos fabris onde certas propriedades podem ser medidas diretamente e produtos podem ser recusados. Um problema dessa abordagem é que apesar de entregar produtos dentro da especificação, não questiona se a especificação atende o cliente.
5. **Abordagem baseada em valor:** vê a qualidade em função de custos e preços, em busca de produtos que fornecem desempenho a um preço aceitável ou conformidade a um custo aceitável.

## 6.2. Dimensões da Qualidade

Qualidade é um conceito multidimensional. Um exemplo bastante simples que podemos usar no Brasil é a avaliação das escolas de samba. Decidir qual a melhor escola é como decidir qual a que apresentou a melhor qualidade. Existem prêmios, como o Estandarte de Ouro, que são dados de uma forma holística, pelo voto na melhor escola. Porém, na competição oficial, o resultado é avaliado em vários quesitos: Enredo, Harmonia,

## *6.2. Dimensões da Qualidade*

Bateria, etc. Assim, para calcular a melhor escola são usadas várias dimensões de qualidade. Garvin (1984), ao mesmo tempo que descrevia as visões de qualidade, propôs também 8 dimensões básicas para um arcabouço para pensar a qualidade de um produto: desempenhos, capacidades, confiabilidade, conformidade, durabilidade, manutenibilidade, estéticas e qualidade percebida.

A visão dimensional evoluiu. Hoje as normas de qualidade ISO e a maioria dos autores adotam a visão de qualidade com dimensões. Isso normalmente também acaba resultando em uma visão hierárquica da qualidade de software. Porém modelos iniciais para Qualidade de Software, como o de McCall, que aparece na Figura 6.3, não eram hierárquicos (McCall, Richards e Walters, 1977). De certa maneira, os modelos hierárquicos são mais simples que os não hierárquicos, porém podem perder certa representatividade. As abordagens atuais também tentam ser, pelo menos minimamente, exaustiva, isto é, apresentam a lista completa de dimensões de qualidade esperadas para um tipo de produto, como Software. Mesmo assim, é claro que produtos específicos podem exigir especializações dos modelos genéricos de qualidade, por exemplo, um modelo de qualidade para software educacional pode exigir mais dimensões ou detalhamento das já existentes para software.

Um modelo de qualidade não é composto apenas pela organização hierárquica de suas dimensões de qualidade, mas também por um forma de pensar que gera o modelo. O Modelo Fuzzy de Qualidade de Software, descrito na Figura 6.4, por exemplo, discute a organização hierárquica a partir de objetivos de software, e dois tipos de relações, as lógicas, que relacionam conceitos, e as quantitativas, que relacionam as formas de agregação do conceito. Além disso, define que existe um nível mensurável na hierarquia de dimensões, o dos critérios, que precisam ter processo de avaliação definido(Belchior, 1997).

Já o modelo de qualidade da norma **ISO 25010**, atualizado no momento da escrita deste texto, é um modelo hierárquico dividido em duas partes, Qualidade do Produto e Qualidade em Uso, representados, respectivamente, pelos mapas mentais das Figuras 6.5 e 6.6(ISO, 2011).

## 6. Qualidade

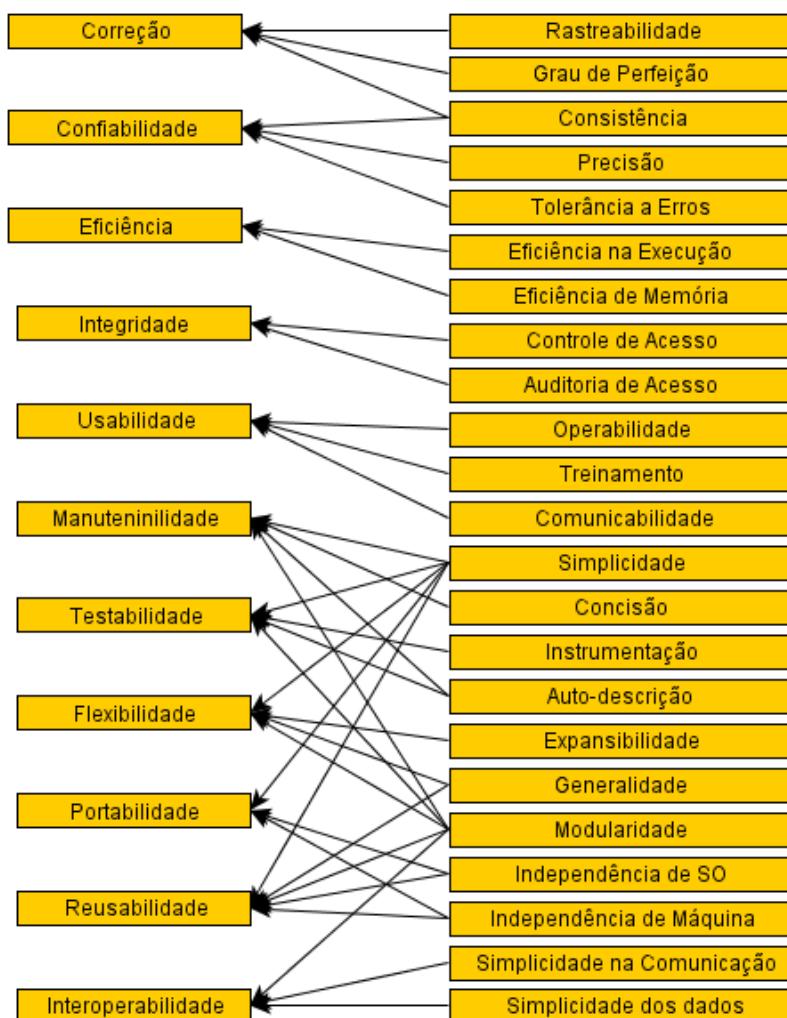


Figura 6.3.: Modelo de Qualidade de McCall. Fonte:(McCall, Richards e Walters, 1977)

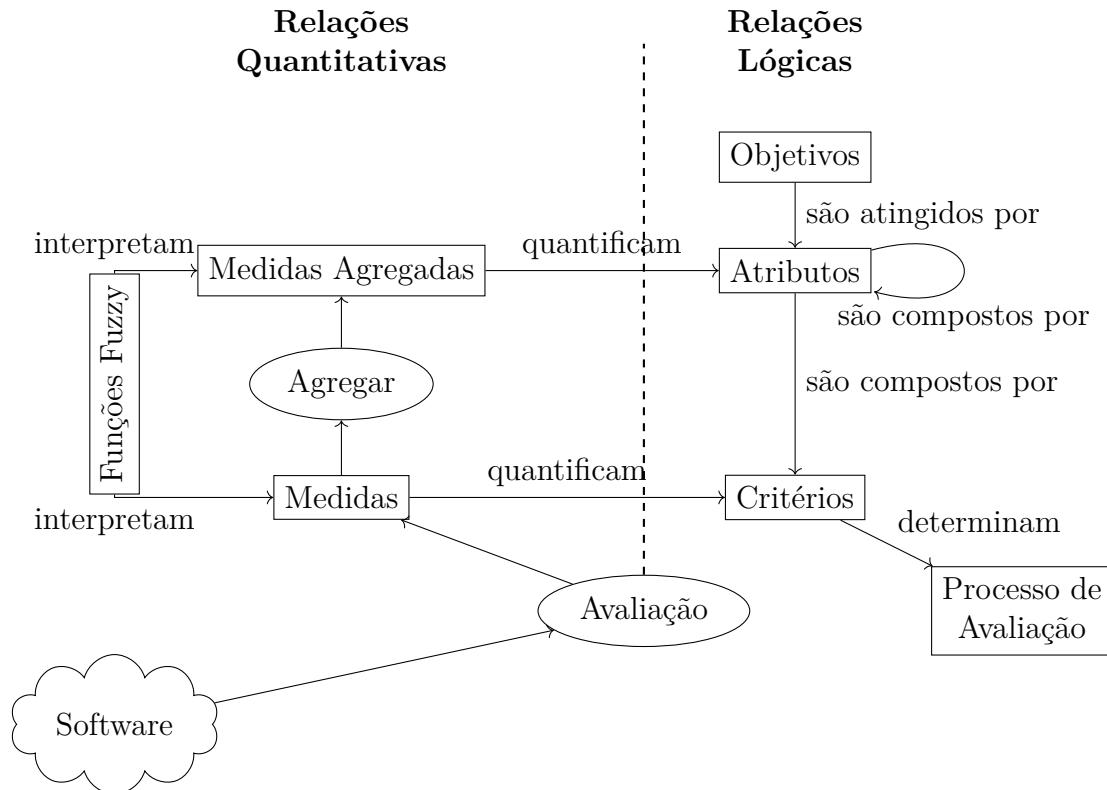


Figura 6.4.: O Modelo Fuzzy de Qualidade de Software. Fonte:(Belchior, 1997)

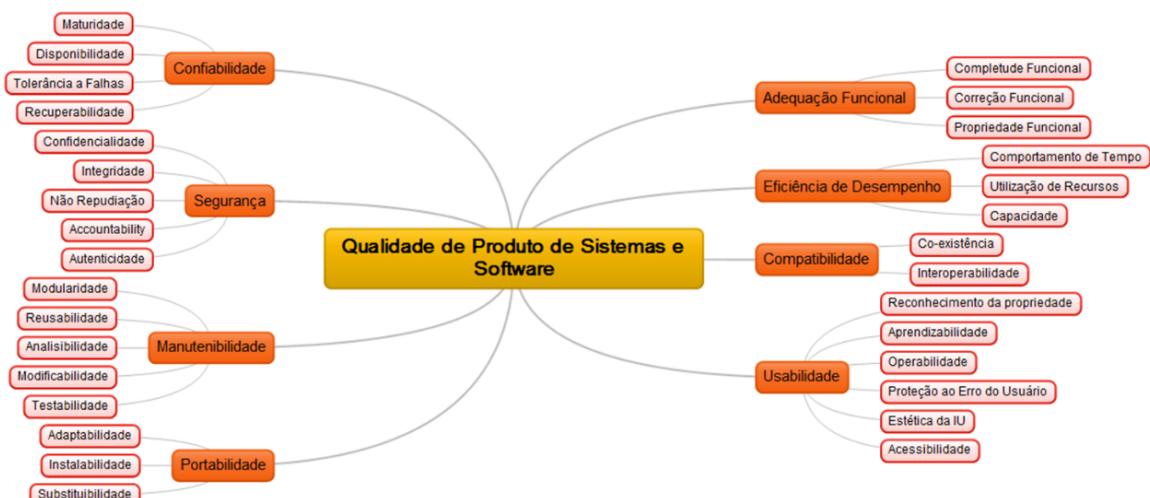


Figura 6.5.: Modelo de Qualidade de Produto de Software da norma ISO 25010.  
Fonte:(ISO, 2011)

## 6. Qualidade

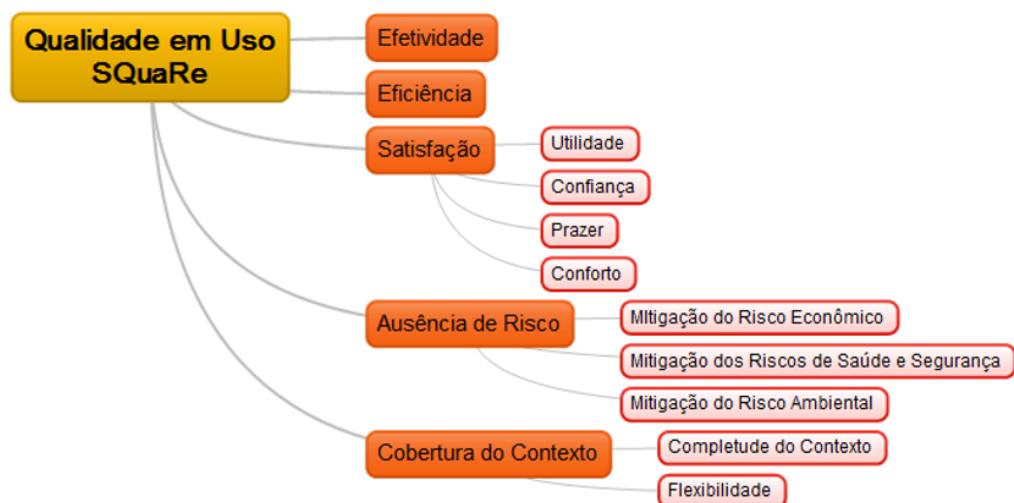


Figura 6.6.: Modelo de Qualidade em uso de Software da norma ISO 25010.

Fonte:(ISO, 2011)

## 6.3. Qualidade é Sempre por um Ponto de Vista

É importante notar que qualidade é um atributo associado a algo (objeto, processo, serviço) por uma pessoa. Assim, diferentes pessoas podem ter diferentes visões de qualidade.

Outra analogia que podemos dar é a da escolha de um lanche por duas pessoas. Vamos supor que Alice e Bob desejam escolher um lanche para a tarde. Eles resolvem então discutir o que esperam do lanche e descobrem 3 dimensões de qualidade: sabor, ser saudável, e praticidade.

O problema de escolher o melhor lanche, isto é, o que tem mais qualidade, é que Alice e Bob tem visões diferentes da importância dessas dimensões. Bob acha o sabor mais importante, enquanto Alice acha que fazer bem a saúde é mais importante.

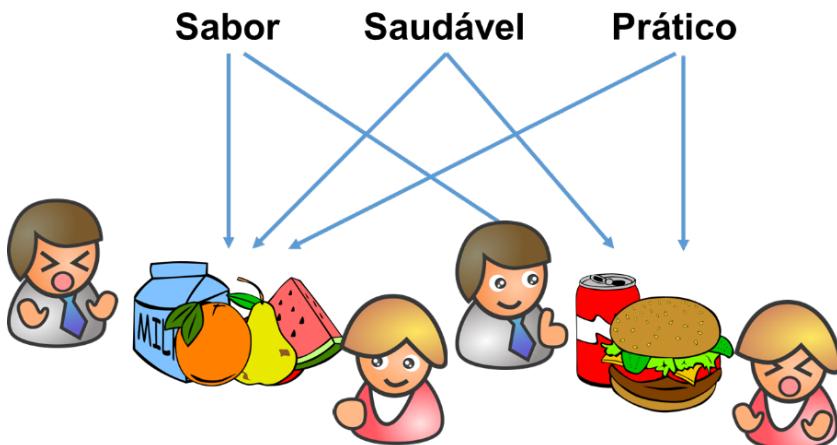


Figura 6.7.: Diferentes pontos de vista implicam em diferentes avaliações de qualidade. Fonte: próprio autor

Da mesma forma, pessoas e organizações diferentes tem visões diferentes de que dimensões são importantes na qualidade de software. Em uma pesquisa realizada na COPPE, por exemplo, verificou-se que a questão mais importante de qualidade de software financeiro é a precisão, composta de correção e acurácia, com a nota “extremamente alta”. Basicamente isso significa que ao construir software financeiro é extremamente importante que as contas estejam corretas. Já a independência de ambiente foi considerada entre “baixa” e “muito baixa”. Imagine agora outro tipo de software, que deve funcionar em qualquer celular, nesse caso a independência de ambiente certamente seria mais necessária (Belchior, 1997) .

## 6.4. Qualidade de Produto e Qualidade de Processo

Uma das principais preocupações da Engenharia de Software é com a qualidade do processo de desenvolvimento de software. Essa preocupação vem de uma premissa básica dos estudos de qualidade: a qualidade do processo é determinante da qualidade do produto.

Essa preocupação é bastante válida, pois tem relação com o importante problema de como garantir a qualidade do produto. A resposta seria: garantindo que o processo que o produz é de qualidade.

Baseado nessa premissa, o Departamento de Defesa Americano solicitou ao Software Engineering Institute da Universidade Carnegie-Mellon para desenvolver um modelo de maturidade de processo que pudesse ser usado como uma medida da qualidade do processo de desenvolvimento de software de uma empresa a ser contratada. Isso resultou no **Capability Maturity Model for Software (CMM)** que mais tarde levou a criação do **Capability Maturity Model Integration (CMMI)**.

O CMMI é uma “coleção de melhores práticas que ajudam organizações a melhorar seus processos” (CMMI Product Team, 2010). Na prática é um modelo que indica quais práticas são necessárias para atingir um processo de alta qualidade. Ele permite que organizações sejam avaliadas e recebam certificações sobre a maturidade de seus processos.

Baseado na ideia do CMMI, o Programa Brasileiro de Qualidade e Produtividade Software decidiu apoiar uma iniciativa similar, denominada MPS.Br, que se apoia no CMMI e em outras normas internacionais.

Além disso, existem normas ISO relacionadas ao processo de software, como por exemplo as normas: ISO 12207:2017 (ISO, 2017), ISO 15504 (ISO, 2004) e ISO 33001:2015 (ISO, 2015a).

## 6.5. Falhas e Defeitos

Falta de qualidade está normalmente associada a falhas na fase operacional. Para discutir essas falhas é necessário ter um vocabulário comum. Esse vocabulário é (também) definido na norma *IEEE Recommended Practice on Software Reliability*.

**Erro (Error)** Uma ação humana que produziu um resultado incorreto, como o software conter uma falta. Exemplos incluem omissões, interpretações erradas de requisitos do usuário ou especificação de software, tradução incorreta ou a omissão de um requisito na especificação de projeto.

**Defeito (Defect)** Um problema que, se não corrigido, pode causar uma aplicação a falhar ou produzir um resultado incorreto.

**Falta (Fault)** (A) Um defeito no código que pode causar uma ou mais falhas, ou (B) a manifestação de um erro no software.

**Falha (Failure)** (A) A inabilidade de um sistema ou componente de sistema de desempenhar uma função requisitada dentro dos limites especificados, ou (B) a terminação da habilidade de um produto de desempenhar um função requisitada ou sua inabilidade de desempenhar dentro dos limites previamente especificados. (C) A saída da operação do programa de dentro dos seus requisitos. Uma *falha* pode ser produzida quando uma falta é encontrada e resulta na perda dos serviços especificados.

Essa definição nos permite entender a ideia que humanos fazem erros, que levam a defeitos e faltas, que causam falhas.

## 6.6. Quanto Custa a Qualidade

Qualidade não é de graça. Para alcançá-la é preciso investir recursos, e se não alcançá-la, será necessário pagar para consertar os defeitos. A experiência mostra que esse investimento compensa, principalmente para grandes projetos.

O SWEBOK (IEEE Computer Society, 2014a) indica os seguintes custos de Qualidade:

- Prevenção
  - Melhoria de Processo
  - Qualidade da Infra Estrutura
  - Treinamento
  - Auditoria
  - Revisões Gerenciais
- Avaliação (*Appraisal*)
  - Achando Defeitos
    - ◊ Revisões
    - ◊ Testes
- Falhas Internas
  - Corrigir defeitos encontrados na avaliação
- Falhas Externas
  - Corrigir defeitos encontrados após o lançamento

C. Jones e Bonsignour (2012) apresenta uma tabela de custos típicos de software por tamanho (em Pontos de Função) e nível de qualidade, replicada na Tabela 6.1. Fica claro que o custo total de desenvolvimento de software é menor quando a qualidade é maior, o que justifica o investimento em qualidade.

O Custo da Qualidade leva a um dilema: qual o investimento que deve ser feito? Se não há nenhum investimento, ninguém vai querer usar o software, se o investimento é em direção a uma qualidade perfeita, o custo pode ser altíssimo. Então é necessário encontrar o ponto mágico, onde é bom o suficiente para não ser rejeitado, mas não tão perfeito que seja inviável(Venners, 2003).

Isso leva há algumas organizações e autores defender o conceito de *Good enough software*.

## 6. Qualidade

Tabela 6.1.: Custos típicos de software por tamanho e nível de qualidade. Fonte:(C. Jones e Bonsignour, 2012)

Pontos de Função	Qualidade		
	Baixa	Média	Alta
10	6,875	6,250	5,938
100	88,561	78,721	74,785
1000	1,039,889	920,256	846,636
10000	23,925,127	23,804,458	18,724,012
100000	507,767,782	433,989,557	381,910,810

## 6.7. Gestão da Qualidade

Tendo em vista que Qualidade tem um custo e precisa ser alcançada de forma ativa, então é necessários haver um processo de gestão da qualidade.

Existem várias normas sobre o assunto, tanto em relação a qualidade em geral como a qualidade específica de software, incluindo gestão e garantia de qualidade.

A norma ISO 12207 (ISO, 2017) propõe um Processo de Gestão de Qualidade, cujo objetivo é “garantir que produtos, serviços e implementações de processos do ciclo de vida atinjam objetivos de qualidade organizacionais e alcancem a satisfação do consumidor”. Os resultados desse processo são:

- definição das políticas e procedimentos de gestão de qualidade;
- definição os objetivos organizacionais de qualidade;
- definição das responsabilidades e autoridades para a gerência de qualidade;
- monitoração do estado de satisfação do consumidor, e
- ações apropriadas são tomadas quando os objetivos de qualidade não são alcançados.

Segundo a norma ISO 9000-2015 (ISO, 2015b), um Sistema de Gestão de Qualidade (SGQ) “compreende atividades pelas quais a organização identifica seus objetivos e determina os processos e recursos necessários para alcançar os resultados necessários”.

O princípios da Gestão de Qualidade são:

**Foco no Cliente** Atender às necessidades e empenhar-se em exceder as expectativas

**Liderança** Líderes estabelecem unidade de propósito, direcionamento e criam condições

para as pessoas engajadas alcançar os objetivos

**Engajamento das pessoas** Pessoas competentes, com poder e engajadas são essenciais para aumentar a capacidade da organização em criar e entregar valor

**Abordagem de processo** Resultados consistentes e previsíveis são alcançados de forma mais eficaz e eficiente

**Melhoria** Foco contínuo na melhoria

**Tomada de decisão com base em evidência** Decisões com base na análise e avaliação de dados e informações são mais propensas a produzir resultados desejados

**Gestão de relacionamento** Para o sucesso sustentado, as organizações gerenciam seus relacionamentos com as Partes Interessadas.

## 6.8. Exercícios

### Exercício 6.1:

Escolha um produto de software qualquer e tente fazer uma análise de sua qualidade.

### Exercício 6.2:

## *6. Qualidade*

Discuta as diferenças de qualidade esperadas entre um jogo vendido em cartucho para um console, um jogo vendido para download no PC e um jogo para *smartphones*.

## **Parte II.**

# **Os Clientes**



# 7

## Oportunidades e Problemas

A wise man will make more opportunities than he finds.

(*Francis Bacon*)

### Conteúdo

---

7.1.	Oportunidades . . . . .	104
7.2.	Problemas . . . . .	105
7.3.	Tipos de Problemas . . . . .	106
7.4.	Identificando Problemas . . . . .	106
7.5.	Caracterização do Problema . . . . .	115
7.6.	Oportunidade e 5W2H . . . . .	116
7.7.	Propondo Oportunidades ao Cliente . . . . .	117
7.8.	Oportunidade e Valor . . . . .	118

## Por que oportunidades e problemas?

Um projeto de software sempre tem como objetivo aproveitar oportunidades ou resolver problemas. Esses termos, porém, devem ser usados de forma mais precisa para que possamos entender os verdadeiros objetivos do projeto, de acordo com as partes interessadas.

Neste capítulo serão discutidas algumas técnicas para descobrir, caracterizar e classificar oportunidades e problemas. As oportunidades são fornecidas pelas partes interessadas, discutidas no Capítulo 5.

## 7.1. Oportunidades

Um **oportunidade** é uma circunstância que torna apropriado o desenvolvimento de um sistema de software(OMG, 2018). Elas são “uma chance de fazer algo, incluindo consertar um problema existente”(Jacobson, Lawson e Ng, 2019).

Uma oportunidade pode ser uma combinação de problemas, sugestões, percepções de mercado ou diretrizes. Ela articula a razão de criação de um sistema novo ou da modificação de um sistema(OMG, 2018). Ruble (1997), que separa oportunidades de problemas, afirma que enquanto problemas sempre indicam que algo está quebrado, em uma oportunidade isso não é necessário.

As oportunidades devem ser entendidas pela equipe de desenvolvimento de forma a gerar os requisitos. Elas justificam e dão foco aos requisitos(OMG, 2018).

As principais oportunidades se originam das partes interessadas que não são a equipe, porém a equipe de desenvolvimento pode propor também oportunidades, ou reconhecer problemas, sendo necessário então ter o acordo das partes interessadas de que devem ser consideradas no projeto. Um cuidado especial deve ser tomado quando a equipe propõe uma oportunidade, que é a tendência de dar importância excessiva a tecnologia, o que pode gerar um requisito falsoMcMenamin e Palmer (1984).

As oportunidades permitem:

- identificar e motivar as partes interessadas;
  - entender o valor que o sistema de software oferece as partes interessadas;
  - entender por que o software está sendo desenvolvido;
  - entender como o sucesso da implantação do sistema de software vai ser julgado e avaliado, e
  - garantir que o sistema de software aborda efetivamente as necessidades de todas as partes interessadas.

É muito importante avaliar continuamente a viabilidade das oportunidades. Jacobson, Lawson e Ng (2019) chamam a atenção para o fato que é necessário:

- confirmar que a oportunidade atende uma **necessidade verdadeira** do cliente;
- tomar decisões sobre que soluções são mais apropriadas para atender a oportunidade, e
- ter a consciência que os benefícios podem demorar algum tempo para ser visíveis para os clientes.

Uma necessidade é verdadeira se o sistema não vai cumprir sua finalidade caso a necessidade não seja atendida(McMenamin e Palmer, 1984).

Oportunidades, positivas ou negativas, são grandes motivadores dos clientes. Se elas não existissem, não haveria necessidade de implementar um novo sistema. A verdadeira motivação de desenvolver o sistema é atender a essas oportunidades, principalmente aqueles que mais afetam o negócio. No *Project Model Canvas* as oportunidades aparecem já no primeiro módulo (Por que), no quadro de justificativas.

## 7.2. Problemas

Um problema é um estado onde existe uma dificuldade que deve ser resolvida, como o desvio de um padrão aceitável, **uma diferença entre o comportamento desejado e o percebido** ou ainda uma causa, real ou possível, de um ou mais eventos que trouxeram ou podem trazer algum prejuízo.

Problemas incomodam as partes interessadas, trazendo prejuízos financeiros, afetando negativamente a imagem da organização, atrapalhando a produção ou os processos, colocando em risco alguma atividade, e mesmo incomodando apenas psicologicamente.

É comum que os problemas apareçam muito rapidamente nas conversas entre as partes interessadas e a equipe de análise, pois provavelmente estão entre os motivos que causaram a necessidade de um novo desenvolvimento de software, porém não necessariamente as partes interessadas têm conhecimento da causa raiz do problema, pois o que as afeta mais diretamente são os sintomas.

Por exemplo, um restaurante pode estar faturando pouco, mesmo com filas na porta, sendo esse o problema que o cliente quer resolver, procurando um novo sistema de pedidos e fechamento de conta. A causa real, porém, pode ser uma cozinha mal planejada, um salão mal aproveitado ou garçons mal treinados. Nesse caso, um novo sistema dificilmente resolveria o problema.

Deve se tomar cuidado para não procurar uma solução sem que o problema esteja claramente determinado. Por isso é importante descobrir, para cada problema, a sua **causa raiz**. Fazê-lo resulta na continuação do problema e do provável abandono da solução.

## 7. Oportunidades e Problemas

Além disso, problemas devem ser definidos em termos de comportamento, não em termos de atitudes ou sentimentos. Para um problema estar realmente descrito, isso deve ser feito na forma de um comportamento que acontece no momento e que é indesejado, colocado em comparação com um comportamento desejado. Segundo Blanchard e S. Johnson (1983), se não há um comportamento desejado não há um problema, mas uma reclamação, e ainda é necessário definir o problema.

### 7.3. Tipos de Problemas

Podemos identificar três tipos básicos de problemas que são descritos pelos clientes:

- **problemas funcionais**, que ocorrem quando um sistema não permite uma funcionalidade que o cliente necessita ou deseja;
- **problemas operacionais**, que ocorrem quando uma funcionalidade funciona de forma errada, e
- **problemas de negócio**, que estão relacionados a manutenção do negócio propriamente dito.

Esses três tipos de problemas se relacionam. Um problema de negócio, por exemplo, pode ser um sintoma da falta de uma funcionalidade adicional no sistema ou de um problema em uma funcionalidade no sistema atual.

### 7.4. Identificando Problemas

Para analisar o problema, é necessário que os usuários:

- Concordem com a definição do problema;
- Entendam as causas do problema, já que o problema por trás do problema pode ser mais importante, sendo que o problema visto inicialmente é apenas um sintoma;
- identifiquem as partes interessadas e usuários;
- definam a fronteira do sistema de solução. e
- identifiquem as restrições impostas à solução

Algumas práticas para identificar problemas são:

- verificar os resultados obtidos atualmente nos processos e compará-los com objetivos da organização ou padrões do mercado;
- observar o comportamento dos colaboradores;
- levantar a opinião de fornecedores, clientes e vendedores, representantes ou distribuidores, da organização.

Quanto aos resultados obtidos, uma análise do processo da organização pode procurar atividades que:

- contém erros;

- são feitas vagarosamente;
- não são mais feitas como definidas em documentos da organização, e
- não são completadas.

Quanto aos colaboradores, é importante verificar se:

- estão desestimulados;
- não conseguem descrever claramente suas responsabilidades e objetivos, e
- pedem demissão com muita frequência.

Além da visão interna da organização, é interessante olhar para os parceiros externos. A partir deles podemos investigar:

- reclamações;
- sugestões de melhorias;
- queda nas vendas, e
- vendas com perdas.

A identificação de problemas às vezes não é fácil se não há um exemplo a comparar. Como saber se seu serviço é lento sem saber como é o serviço da concorrência? Existem práticas específicas para isso, como a compra de relatórios feitos por consultorias especializadas, *benchmarkings*(Stapenhurst, 2009), etc.

Quando um problema é identificado, é importante que todos concordem que esse é o problema a ser resolvido pelo projeto. No Capítulo ?? é apresentado um método prático para iniciar um projeto com esse acordo, o *Project Model Canvas*.

Para estabelecer o problema, é interessante criar uma sentença apropriada, como a seguinte:

O problema de <*descrição do problema*>, afeta <*partes interessadas afetadas*> e resulta em <*impacto nas partes interessadas*>. A solução <*indicar a solução*> Trará os benefícios de <*lista dos principais benefícios*>.

#### 7.4.1. A Análise de Pareto

Ao lidar com uma quantidade de problemas detectados em um sistema funcionando, é importante determinar os principais problemas, aqueles que tem mais impacto, de modo a dar foco aos esforços e obter melhores resultados.

Uma das técnicas mais conhecidas para isso é a Análise de Pareto.

**O Princípio de Pareto** diz que 20% dos problemas causam 80% dos efeitos. Essa divisão não é real, mas é uma forma de indicar que poucas causas são responsáveis pelos maiores impactos e que é necessário colocar os problemas em perspectiva.

Então, segundo o Princípio de Pareto, devemos calcular o impacto de cada tipo de erro em nosso sistema, selecionando os de maior impacto para resolver.

## 7. Oportunidades e Problemas

Devemos tomar cuidado, porém, porque o resultado que obteremos depende dos dados que tivermos<sup>1</sup>. A seguir são apresentados dois exemplos, baseados na contagem e custo de erros obtidos em um sistema fictício, onde no primeiro apenas são contados apenas os erros por tipo, chegando a uma análise errada do que é importante, e no segundo é calculado o impacto de cada tipo.

### 7.4.2. Calculando a Quantidade

Para tratar problemas de acordo com a técnica de Pareto, **supondo que todo problema tem o mesmo efeito**, é necessário seguir os seguintes passos:

1. dividir os problemas em categorias;
2. levantar o número de ocorrências por categoria;
3. classificar as categorias em ordem decrescente por quantidade de ocorrências, como na Tabela 7.1;
4. calcular o total de ocorrências;
5. calcular a porcentagem de cada categoria;
6. determinar uma escala;
7. colocar as categorias como colunas, ordenadamente, a maior a esquerda, e
8. traçar as curvas de porcentagem acumulada, como na Figura 7.1.

A Tabela 7.1 apresenta o primeiro resultado útil, isto é, a tabela ordenada em ordem decrescente da quantidade de ocorrência por tipo de problema, para problemas encontrados em um sistema fictício.

Tabela 7.1.: Uma contagem de tipos de erros para uso da técnica de Pareto

Problema	Ocorrências
Requisito Original Errado	35
Erro de Programação	25
Erro de Projeto	21
Erro de Entrada de Dados	18
Erro de Formato	13
Problema de Segurança	12
Mudança no Negócio	10
Outros (1 Ocorrência)	7
Mudança na Legislação	5
Falta de Rede	3
Falta de Energia	1

A partir da Tabela 7.1 é possível gerar o gráfico da Figura 7.1, que já permite comparar a importância dos problemas, porém ainda considerando que todos os problemas tem o mesmo impacto, o que é raro acontecer.

<sup>1</sup>Os americanos têm um ditado: *Garbage in, garbage out.*

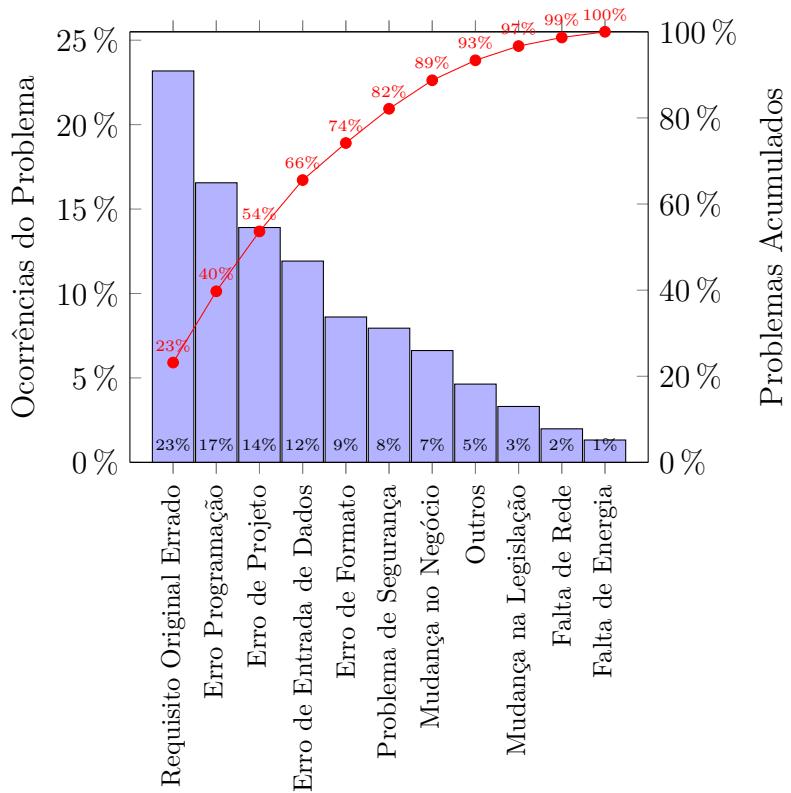


Figura 7.1.: Gráfico com a quantidade de problemas.

A análise da Tabela 7.1 e do Figura 7.1 levaria a crer que 82% dos problemas seriam causados por 6 tipos: Requisitos Originais Errados, Erro de Programação, Erro de Projeto, Erro de Entrada de Dados, Erro de Formato e Problema de Segurança. Na prática, não houve um atendimento do Princípio de Pareto, isso porque estamos apenas contando, e não calculando o impacto. No mundo real, é difícil que a premissa que foi feita nesse cálculo seja verdade, dificilmente todo problema tem o mesmo impacto.

### 7.4.3. Usando o Impacto dos Problemas

A análise anterior pode ser muito melhor se soubermos o impacto de cada problema, seja ele o custo financeiro ou apenas um valor relativo. Dessa forma podemos melhorar a Técnica de Pareto. Vamos supor que os problemas detectados no sistema tenham um custo médio relativo como descrito na Tabela 7.2;

## 7. Oportunidades e Problemas

Tabela 7.2.: Custo médio relativo dos problemas

Problema	Ocorrências
Requisito Original Errado	100
Mudança no Negócio	80
Mudança na Legislação	80
Erro de Projeto	10
Problema de Segurança	10
Outros (1 Ocorrência)	10
Erro de Programação	2
Erro de Entrada de Dados	1
Erro de Formato	1
Falta de Rede	1
Falta de Energia	1

Quando os problemas tem impacto diferente, é necessário multiplicar a quantidade de problemas de um tipo pelo custo médio do problema, gerando uma tabela de impacto dos erros, como na Tabela 7.3, e o gráfico ficaria como na Figura 7.2.

O novo passo a passo, e o mais aconselhado, para aplicar o Princípio de Pareto com a inclusão do impacto de cada tipo de problema é:

1. dividir os problemas em categorias;
2. levantar o número de acontecimentos por categoria, como na Tabela 7.1;
3. levantar o custo médio por categoria, como na Tabela 7.2;
4. multiplicar o custo médio pelo número de acontecimentos;
5. classificar as categorias em ordem decrescente por custo médio, como na Tabela 7.3;
6. calcular o custo total dos problemas;
7. calcular a porcentagem de cada categoria;
8. determinar uma escala;
9. colocar as categorias como colunas, ordenadamente, a maior a esquerda, e
10. traçar as curvas de porcentagem acumulada, como na Figura 7.2.

Analizando esses resultados vemos a importância de calcular não só a quantidade de erros, mas também seu custo, para melhor priorizar os problemas.

Na Figura 7.2 fica claro que apenas três, ou mesmo dois, problemas precisam atenção, sendo que o primeiro problema é bem mais importante que os outros. Nesse caso, o Princípio de Pareto é atendido.

Tabela 7.3.: Custo total relativo

Problema	Ocorrências
Requisito Original Errado	3500
Mudança no Negócio	2000
Mudança na Legislação	1680
Erro de Projeto	180
Problema de Segurança	130
Outros (1 Ocorrência)	120
Erro de Programação	20
Erro de Entrada de Dados	7
Erro de Formato	5
Falta de Rede	3
Falta de Energia	2

#### 7.4.4. Buscando Causas

Um problema é gerado por várias causas, que podem também ser geradas por outras causas. Descobrir a causa raiz de um problema é um processo de investigação que pode ser documentado em um **Diagrama de Espinha de Peixe**, com o da Figura 7.3, também conhecido como **Diagrama de Causa Raiz**.

Esse diagrama é simples de desenhar ou construir, mas certamente esconde toda uma dificuldade de levantar corretamente as informações que revela. Como muitas propostas de análise, o processo de construção, e o aprendizado dele decorrente, é muitas vezes mais importante que o registro completo final.

O Diagrama de Espinha de Peixe indica as causas para um problema de maneira hierárquica. Para cada causa pode haver outras causas, e assim sucessivamente. O objetivo é a partir de um problema achar as causas raiz.

Ele parte da premissa que normalmente problemas são percebidos por um sintoma, e que o verdadeiro problema está escondido em uma camada de problemas sucessivos.

O processo de criação é mais produtivo se feito por um conjunto de pessoas, em uma reunião semelhante a um *brainstorm*. Nesse caso é importante que todos concordem com o efeito ou problema inicial a ser discutido. Sucessivamente devem ser feitas perguntas sobre “por que” o problema ocorre, ou seja, quais suas causas.

Cada linha de causalidade deve ser seguida até sua causa raiz. Isto é, a prática mais recomendada é trabalhar em profundidade, com perguntas sucessivas sobre por que algo acontece. Isso é similar a técnica dos 5 porquês, usada da mesma forma.

Apesar de no início o diagrama poder ser construído de forma livre, é interessante organizá-lo de alguma maneira para a apresentação. Ramos lotados podem ser divididos, ramos quase vazios podem ser unificados. Também é necessário ter uma noção da

## 7. Oportunidades e Problemas

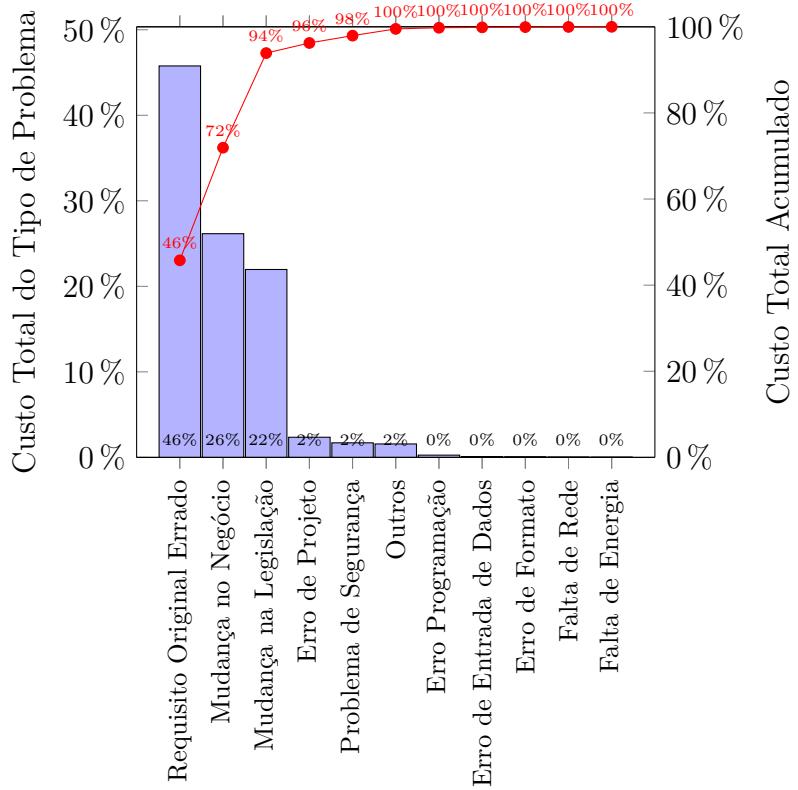


Figura 7.2.: Pareto com custo

importância das causas, porque não é necessário gastar muito tempo em uma causa que não é realmente impactante para o problema.

A construção pode ser feita em um quadro branco ou com um projetor. É importante que todos os participantes tenham acesso completo ao quadro.

O diagrama de Figura 7.3 mostra a investigação de causas para o fato de um restaurante ter longas filas na porta. Pode ser visto que algumas causas conseguiram uma hierarquia de causa raiz mais longa do que outras. Isso não é um problema.

Algumas ferramentas permitem trabalhar com diagramas diferentes mas com a mesma finalidade. Ferramentas de *mind mapping*, por exemplo, permitem não só criar diagramas equivalentes, como permitem também trabalhar com resumos das figuras, como mostrado nas Figuras 7.4 e 7.5

Também é possível usar o diagrama como uma ferramenta pessoal de auxílio a investigação, ou para apoiar a técnicas dos 5 Porquês. Na Figura 7.6, o diagrama é usado para ilustrar o seguinte raciocínio.

1. Tirei uma nota baixa
2. Por que?
3. Porque a prova estava difícil.
4. Por que?

## Problemas do Restaurante

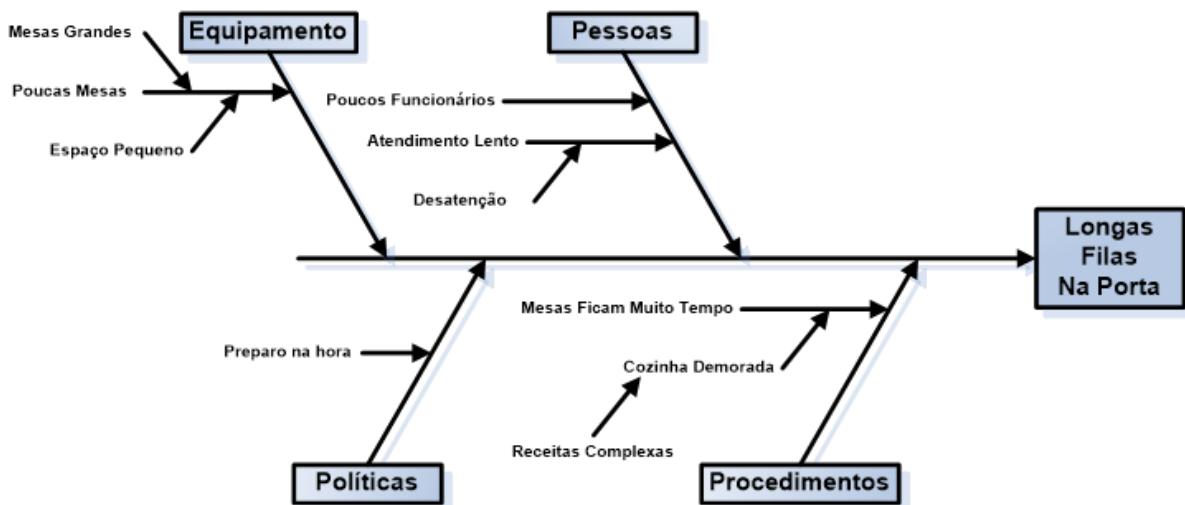


Figura 7.3.: Um diagrama de Espinha de Peixe



Figura 7.4.: Um diagrama de Espinha de Peixe resumido, construído com uma ferramenta de *mind map*.

5. Porque não estudei.
6. Por que?
7. Por que não me organizei.
8. Por que?
9. Acho que não há um motivo para isso.
10. Há outro motivo para tirar uma nota baixa?
11. Sim, não fiz a AD, preparatória para a prova.
12. Por que?
13. Não tive tempo.
14. Por que?
15. Novamente, não me organizei.
16. Por que?
17. Acho que novamente não tenho uma explicação para isso.

## 7. Oportunidades e Problemas

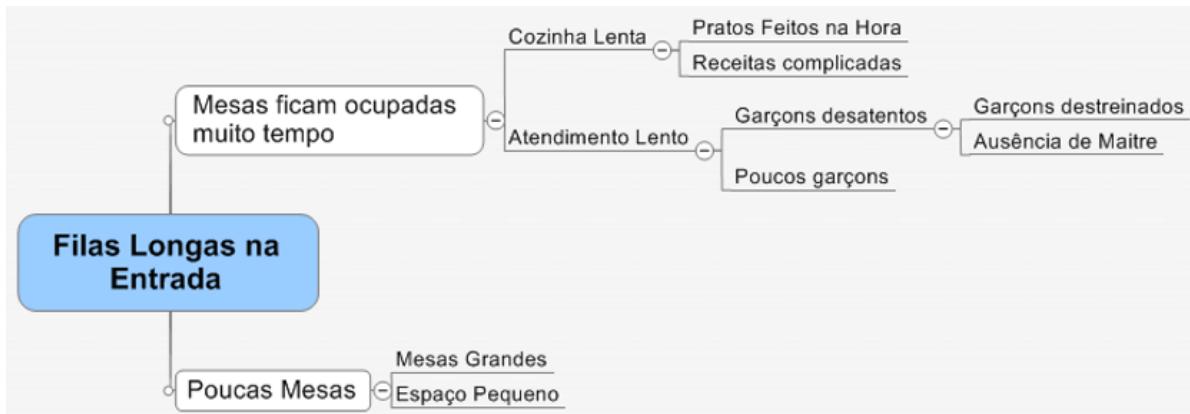


Figura 7.5.: Um diagrama de Espinha de Peixe expandido, construído com uma ferramenta de *mind map*.

### 7.4.5. Organização do Diagrama Espinha de Peixe

O Diagrama de Espinha de Peixe pode ser organizado com as causas agrupadas em categorias. Existem pelo menos 3 organizações reconhecidas: para indústrias, para serviços e administração, e para Gestão de Qualidade Total.

As classes para indústrias são:

- pessoal;
- métodos;
- materiais, e
- máquinas.

As classes para serviços e administração são:

- pessoal;
- equipamentos;
- procedimentos, e
- políticas.

As classes para Gestão de Qualidade Total (TQM, *Total Quality Management*) são:

- ambiente;
- gestão;
- pessoas;
- materiais;
- equipamento, e
- processo.

Finalmente, o método 6-sigma, sugere usar 6 categorias, chegando a chamar o diagrama de 6-M, por usar nomes que começam por M:

- métodos;
- máquinas;

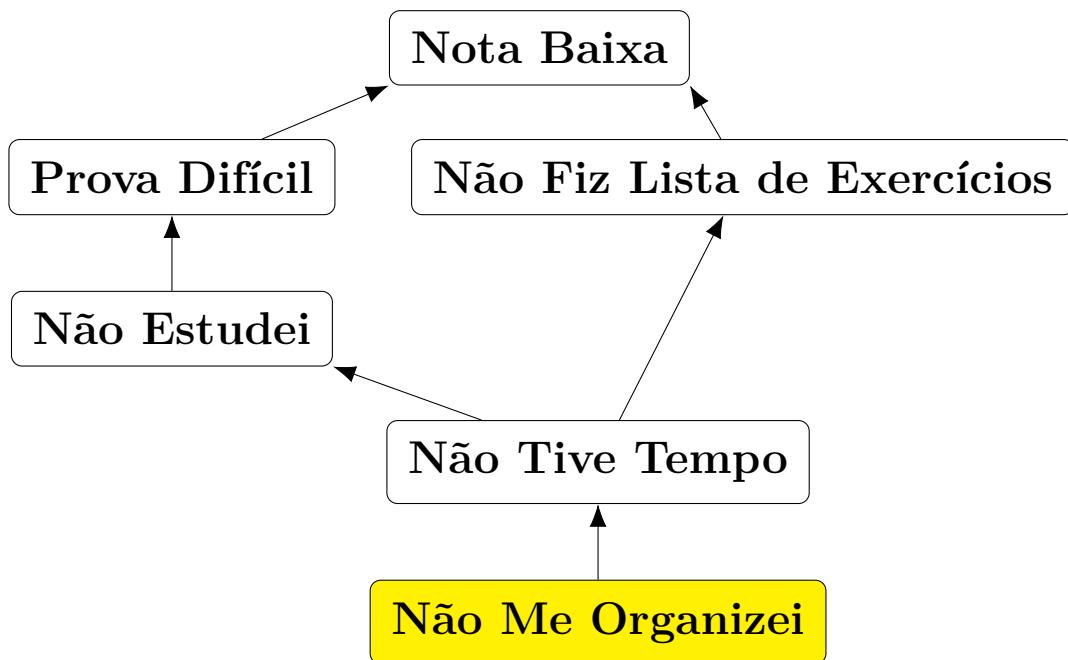


Figura 7.6.: Um Diagrama de Causa Raiz vertical, onde a causa raiz está marcada.

- materiais;
- mão-de-obra;
- meio-ambiente, e
- medida.

Alguns autores ainda incluem outros M, como *management*, usando a tradução forçada de manejo administrativo, e meios financeiros.

Todas essas categorias podem ser usadas nas reuniões como inspiração para a busca de causas, fazendo perguntas como “Existe alguma causa de gestão para esse problema?”.

Mesmo usando essas técnicas, podemos encontrar fatores que não se alinharam às categorias pré-determinadas, ou criam dúvidas sobre em que categoria colocar. Um dos motivos é que elas olham para dentro da organização, e o problema pode ser fora. Pessoas que não querem se vacinar, por exemplo, não seriam encontradas usando apenas o 6M, mas seriam encontradas usando o “Pessoas” em serviço. O importante é entender que essas categorias estão aí para ajudar e não para atrapalhar. Elas nos lembram de onde procurar problemas, e não são uma camisa de força. Se ficar em dúvida de onde colocar algo, coloque em qualquer lugar.

## 7.5. Caracterização do Problema

Identificado o problema é necessário caracterizá-lo claramente. Para isso, é necessário identificar:

## 7. Oportunidades e Problemas

- o problema;
- quem é afetado pelo problema;
- seu efeito no negócio e nas partes interessadas;
- o impacto financeiro;
- suas causas e sua causa raiz;
- seu causador, seja ele um processo, um dado, um profissional, etc.;
- pelo menos uma sugestão de solução, e
- os benefícios que uma solução pode trazer.

Alguns desses dados podem ter sido levantados de forma aproximada na identificação do problema. Nesta fase é importante detalhá-los e documentá-los.

### 7.6. Oportunidade e 5W2H

As oportunidades, ou problemas, bem identificadas devem responder as perguntas 5W2H. Perguntas comuns a serem respondidas são:

- Por que são necessárias? Por que acontecem?
- O que elas são? Qual a causa raiz? Qual o efeito na empresa?
- A quem atendem? A quem incomodam?
- Quando elas ocorrem, quando elas devem ser aproveitadas?
- Quanto elas custam? Quanto elas permitem lucrar?
- Onde ocorrem? Onde podem ser resolvidas ou aproveitadas?

#### 7.6.1. Buscando Oportunidades com as Partes Interessadas

A melhor maneira de levantar oportunidades é entender simultaneamente: o que o cliente deseja, o que está funcionando agora, quais os problemas atuais e quais oportunidades existem para um novo sistema.

As entrevistas iniciais para levantamento de necessidades de informação são geralmente feitas com executivos. Um bom conjunto inicial de questões que podem ser feitas é apresentados na lista a seguir (Gillenson e Goldberg, 1984).

- Quais seus objetivos?
- Quais suas responsabilidades?
- Que medidas você usa?
- Que informações você precisa?
- Quais são seus problemas de negócio?
- Que mudanças você vê no futuro (que vão impactar a infra-estrutura do seu negócio)?
- Quais são os fatores críticos de sucesso?
- Qual a informação mais útil que você recebe?

## 7.7. Propondo Oportunidades ao Cliente

- Como você classificaria as informações que recebe quanto à adequação, validade, duração, consistência, custo, volume, etc.?

Alguns pontos podem ser notados sobre essas perguntas. A pergunta sobre objetivos muitas vezes pode não ser muito bem respondida, assim a segunda pergunta, sobre responsabilidades, permite uma resposta mais prática e mais fácil de ser dada.

A terceira pergunta visa buscar uma compreensão sobre os dados importantes para o entrevistado, abrindo uma sequência de perguntas sobre o tema.

A quinta pergunta, sobre os problemas de negócio, é a mais importante do conjunto e deve ser dada a maior atenção e quantidade de tempo disponível.

É interessante que o problema seja estruturado em um formato causa-efeito, por exemplo: "por causa da falta de dinheiro, o resultado é que não é possível comprar peças de reposição". Também é importante verificar se a causa primordial do problema é um processo ou um dado.

Além disso, o entrevistador deve tentar obter alguma informação de valor (financeiro) sobre o impacto do problema, seja em valores absolutos ou relativos, objetivos ou subjetivos.

## 7.7. Propondo Oportunidades ao Cliente

O analista de sistemas, com sua experiência tecnológica e de negócio, também pode oferecer oportunidades aos seus clientes.

Uma **oportunidade tecnológica**, como diz o nome, é uma tecnologia específica a ser usada na implementação e que oferecerá alguma vantagem ao cliente. Por exemplo, ao implantar um novo sistema de estoque podemos oferecer a entrada e saída de produtos pelo uso de código de barras. A oportunidade tecnológica não altera a funcionalidade que o cliente necessita, registrar a entrada de um item no estoque, mas sim sua forma de implementação.

Algumas vezes também podemos oferecer **oportunidades de negócio**.

Devemos ter muito cuidado ao propor oportunidades, pois podemos estar criando falsos requisitos, discutidos no Capítulo 8. Um exemplo típico de falso requisito, que costuma ser proposto por analistas, é a proliferação de relatórios em sistemas que na prática usam apenas alguns.

Uma oportunidade de negócio deve ser exaustivamente discutida com o cliente de forma a ficar claro que ela realmente traz benefícios, que esses benefícios são consideráveis, que o risco do projeto não aumenta muito e que ela será realmente utilizada.

Um exemplo que podemos dar é, em um controle de estoque, a funcionalidade de prever que produtos estão próximos de sua data de vencimento. Essa não é uma função "normal" de sistemas de estoque. Exige um custo adicional não só de desenvolvimento, mas também

## *7. Oportunidades e Problemas*

de operação, como identificar a data de vencimento, controlar diferentes datas para um produto, etc. Em muitos contextos, essa função pode parecer interessante mas ser, na prática, inútil. Em uma oficina mecânica, por exemplo, ela pode ser totalmente inútil. Em um grande mercado, ela pode ser bastante útil. Porém em um pequeno mercado, onde o proprietário tem na verdade um controle mental do estoque e precisa do sistema de estoque apenas para fazer um controle legal ou financeiro, essa funcionalidade pode parecer útil a princípio, mas nunca ser usado no dia a dia.

## **7.8. Oportunidade e Valor**

As oportunidades devem produzir resultados úteis para a empresa, como acelerar processos, eliminar passos desnecessários, reduzir erros na entrada e saída de dados, aumentar a integração entre sistemas, aumentar a satisfação do usuário e facilitar a interação com os parceiros externos da organização (fornecedores, representantes e clientes)(K. E. Kendall e J. E. Kendall, 2013).

A prática IRACIS ajuda a buscar oportunidades, ou a analisar se algo é ou não uma oportunidade, que é tratada na Seção ??, busca encontrar valor por meio de aumentar as receitas, diminuir os custos e melhorar os serviços.

Já os elementos de valor, tratados na SubSeção 3.3.1, permitem também buscar ou analisar outras formas de valor, tanto para pessoas físicas, em sistemas B2C, como para pessoas jurídicas, em ambientes B2B.

Claramente, uma oportunidade (problema) precisa de um valor associado a ela para que se possa reivindicar o seu aproveitamento (solução). Quanto mais este valor puder ser resumido em um valor financeiro, mas fácil é verificar se vale a pena investir em um sistema de software para aproveitá-la (resolvê-lo). Calcular o impacto financeiro de uma oportunidade ou problema é responsabilidade da área de negócio e não do analista de sistemas. Porém, cabe ao analista analisar se o benefício pretendido é factível com o projeto sendo proposto.

Para fazer a decisão financeira final podem ser usadas as técnicas ROI, IRR e NPV apresentadas na Seção 3.5.

## **7.9. Exercícios**

### **Exercício 7.1:**

Associe as características que devem ser levantadas para um problema às perguntas 5W2H.

### **Exercício 7.2:**

Analise o local onde você estuda e os serviços de software fornecidos a você. Faça uma lista de problemas e oportunidades. Discuta as causas raiz dos problemas.

**Exercício 7.3:**

Vá para o site <http://jogodeanalisedesistemas.xexeo.net/> e visite a Livraria Resolve. A partir da sua visita liste os problemas e oportunidades.



# 8

## Requisitos

The hardest part of the software task is arriving at a complete and consistent specification, and much of the essence of building a program is in fact the debugging of the specification.

*(Frederick P. Brooks)*

### Conteúdo

---

8.1.	Definição Formal . . . . .	122
8.2.	Requisitos x Benefícios . . . . .	126
8.3.	Tipos de Requisitos . . . . .	126
8.4.	O Papel dos Requisitos no Projeto . . . . .	128
8.5.	Engenharia de Requisitos . . . . .	129
8.6.	Elicitação de Requisitos . . . . .	129
8.7.	Níveis de Abstração dos Requisitos . . . . .	134
8.8.	O Risco de Requisitos Falsos . . . . .	134
8.9.	Características de um Bom Requisito . . . . .	135
8.10.	Atributos dos Requisitos . . . . .	137
8.11.	Priorizando Requisitos . . . . .	139

## 8. Requisitos

8.12.	Requisitos Mudam com o Tempo . . . . .	140
8.13.	Requisitos e Necessidades . . . . .	141
8.14.	Requisitos de Acordo com Normas IEEE . . . . .	142
8.15.	A Visão do Novo Sistema . . . . .	145
8.16.	Descrevendo Requisitos . . . . .	145
8.17.	Exercício . . . . .	147

### Por que requisitos?

Levantar e especificar os requisitos são a principal função da análise de sistemas, seja qual for a metodologia usada.

Para construir qualquer coisa é preciso saber o que vai ser feito, de maneira a atender da melhor forma possível as demandas das partes interessadas. No caso de software, ou de sistemas em geral, este conhecimento normalmente é registrado na forma de uma **especificação de requisitos**. Em uma interpretação restrita, requisito de software é uma exigência feita ao software no início do projeto.

Como projetos de software normalmente são desenvolvidos em fases, sejam elas sequenciais ou cíclicas, é comum que uma fase anterior construa artefatos de conhecimento que permitem definir melhor o que será feito na próxima fase. Todo artefato criado em uma fase de projeto que define de alguma forma o que é preciso ser feito em uma fase posterior de um projeto pode ser chamado também de requisito<sup>1</sup>. Assim, em uma interpretação ampla, qualquer especificação, realizada em uma fase qualquer do processo de software, é um requisito para a próxima fase, já que ela tem que ser cumprida.

Essa ambiguidade, entre a interpretação ampla e restrita, normalmente é resolvida pelo contexto. Outros termos também são adicionados para qualificar o termo requisito e ajudar na interpretação do seu significado exato. Por exemplo, requisitos do usuário é o termo usado normalmente para significar requisitos levantados pelo usuário, com ou sem ajuda de analistas de sistemas, em uma fase anterior, ou no início, da análise de sistemas.

### 8.1. Definição Formal

Segundo a norma *ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary*(IEEE, 2010), um **requisito** é:

- uma condição ou capacidade necessária para um usuário resolver um problema ou atingir um objetivo;

---

<sup>1</sup>O termo em inglês é *requirement*, o que levou a algumas traduções usarem o termo requerimento, que é errado em português.

- uma condição ou capacidade que precisa ser possuída ou alcançada por um sistema, produto, resultado, serviço ou componente para satisfazer um acordo, contrato, padrão, especificação ou outro documento formalmente imposto, ou
- Uma representação documental de uma condição ou capacidade no sentido das duas definições anteriores.

Esta definição, múltipla, permite o uso da interpretação ampla de requisitos.

Sommerville e Sawyer (1997) definem requisitos como

uma especificação do que deve ser implementado. Eles são descrições de como o sistema deve se comportar, ou de uma propriedade ou atributo do sistema. Eles podem ser uma restrição ao processo de desenvolvimento do sistema.

Esses autores também afirmam que “não existe uma melhor forma de escrever requisitos”, isto é, cada projeto pode precisar de uma forma específica de acordo com condições como assunto, cultura organizacional, questões de segurança, etc. Por exemplo, é razoavelmente mais caro fazer especificações em linguagens formais, exigindo pessoal com mais formação, software especializado, etc. Além disso, especificações formais são difíceis de serem validadas ou negociadas com o cliente, logo raramente são usadas em desenvolvimento de sistemas comerciais. Porém são usadas em especificações de sistemas que precisam de níveis adicionais de garantias de segurança e robustez, como o *TCAS* (*Traffic Alert and Collision Avoidance System*) e outros (Craigen, Gerhart e Ralston, 1993).

### 8.1.1. Exemplos de Requisitos

Como visto, a noção de requisito pode ser muito ampla. Eles podem ser encontrados em várias formas, dependendo dos métodos e práticas usados em cada projeto de software específico.

Na sua forma mais tradicional, um requisito é apenas uma sentença que define uma exigência feita ao sistema, na perspectiva do usuário, escrita em linguagem corrente, como em:

**“O sistema deverá permitir que o correntista verifique o saldo de sua conta”.**

Em casos de requisitos escrito em linguagem natural, muito encontrados em documentos conhecidos como **RFP**, **Request for Proposal**, a forma mais correta é que a exigência seja sempre feita ao sistema sendo especificado, nunca ao usuário, no caso o correntista, porém não é incomum encontrar RFPs contendo erros de perspectiva, já que muitas vezes são construídas por pessoas leigas em relação a área de Engenharia de Software.

Muitas formas já foram propostas para eliciar ou analisar requisitos, ou para fazer uma parte desse trabalho. Algumas deram foram:

- eventos essenciais (McMenamin e Palmer, 1984);
- diagramas de fluxo de dados (DFD) (Gane e Sarson, 1979);
- mini especificações (Gane e Sarson, 1979);

## 8. Requisitos

- diagramas de contexto (Gane e Sarson, 1979; K. E. Wieggers e Beatty, 2013);
- cartões volare, como o da Figura 8.1 (J. Robertson e S. Robertson, 2006);
- especificações formais(Sommerville, 2015);
- casos de uso (Jacobson, Christerson et al., 1992; Jacobson, Spence e Bittner, 2011);
- diagramas de caso de uso, como o da Figura 8.2 (Jacobson, Christerson et al., 1992; OMG, 2017);
- mapas mentais, como o da Figura 8.3;
- descrições em linguagem natural e hierárquicas (K. E. Wieggers e Beatty, 2013)
- árvores de funcionalidades (K. E. Wieggers e Beatty, 2013), e
- histórias do usuário (Cohn, 2004).

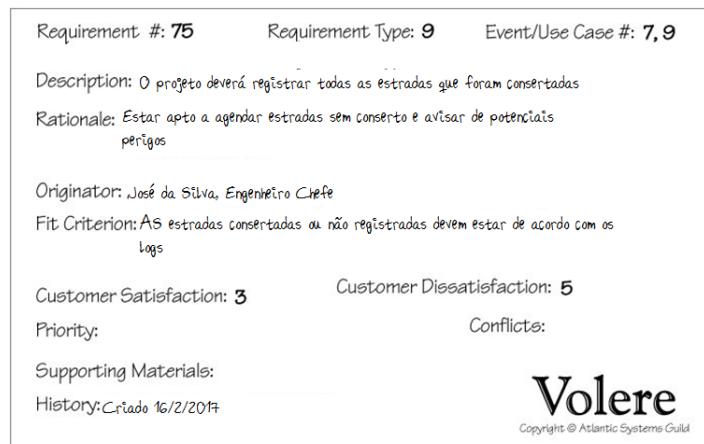


Figura 8.1.: Um cartão Volere.

As duas formas mais em voga no mercado em 2020 serão apresentadas nos capítulos a seguir: histórias do usuário e casos de uso.

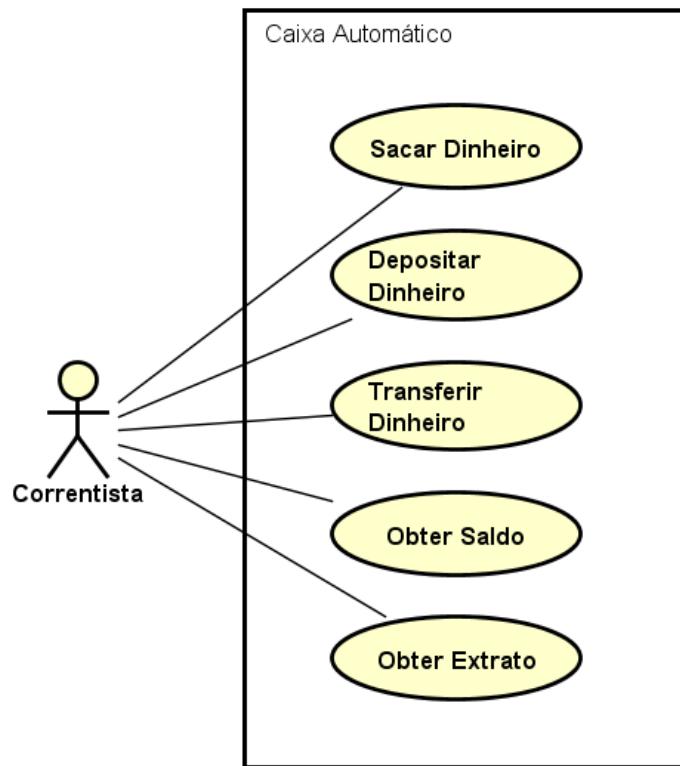


Figura 8.2.: Exemplo de um diagrama de casos de uso

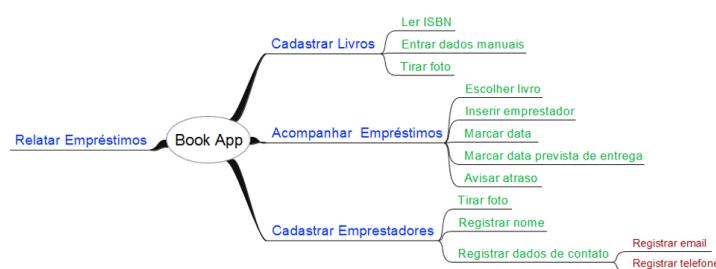


Figura 8.3.: Exemplo de um mapa mental.

## 8.2. Requisitos x Benefícios

No Capítulo ?? apresentamos brevemente requisitos e benefícios. Um benefício típico seria “diminuir os erros no preenchimento dos pedidos”, enquanto um requisito típico seria “o sistema deve oferecer ao comprador a lista de produtos disponíveis na loja”.

Algumas pessoas podem pensar que um benefício também é algo que se pede ao software, logo também é um requisito. Isto está errado. Um requisito é sempre uma demanda determinada diretamente para o produto ou serviço sendo especificado, enquanto um benefício é uma mudança esperada no negócio quando o produto ou serviço estiver em operação. Não se pode testar um benefício, por exemplo, a não ser que haja, para ele, um requisito específico em paralelo. Um exemplo disso seria um benefício como “permitir compra com cartão de crédito” e um requisito na forma “o sistema deverá permitir que o cliente pague com cartão de crédito”.

## 8.3. Tipos de Requisitos

Vários autores dividem os requisitos de formas diferentes, porém quase todos concordam que existem dois tipos básicos de requisitos: os **requisitos funcionais** e os **requisitos não funcionais**. Além disso é comum que um tipo de requisito seja colocado a parte, as **restrições**.

### 8.3.1. Requisitos Funcionais

Um requisito funcional representa algo que o sistema deve fazer, ou seja, uma função esperada do sistema que agregue algum valor a seus usuários. Exemplos típicos incluem a emissão de relatórios e a realização e manutenção de cadastros.

Um requisito funcional tradicional é escrito de uma forma específica, exigindo que o sistema cumpra uma finalidade, como em “o sistema deve permitir que um aluno solicite a emissão do seu histórico escolar”. Também é possível que um requisito funcional seja uma especificação mais detalhada, impondo limites ou exigências a outro requisito, com em: “O CR será calculado a partir da média ponderada das notas em cada curso, sendo que o peso de cada nota será a quantidade de créditos acreditada ao curso.”

### 8.3.2. Requisitos não funcionais

Requisitos não funcionais falam da forma como os requisitos funcionais devem ser alcançados. Eles definem propriedades e restrições do sistema. Muitos requisitos não funcionais são também requisitos de qualidade, como exigências de desempenho e robustez. Outros são restrições ou exigências de uso de uma ou outra tecnologia.

### 8.3. Tipos de Requisitos

Muitas vezes não é só difícil descobrir quais são os requisitos não-funcionais, mas também produzir uma especificação do sistema que possa ser cumprida em custo razoável e prazo hábil de forma a atender os usuários. Um exemplo típico seriam dois requisitos não funcionais que, genericamente, são opostos: velocidade e transportabilidade. Ora, para fazer um software muito veloz você precisa adaptá-lo especificamente para o ambiente onde ele está funcionando. Para fazer um software transportável, você precisa implementá-lo de forma a funcionar no maior número possível de ambientes. Normalmente esses dois requisitos se relacionam de forma inversa e para implementá-los simultaneamente é necessário um grande investimento de recursos.

Alguns requisitos não funcionais, quando negados, geram um “anti-requisito” que nunca seria pedido por um cliente. Por exemplo, um requisito não funcional comum é que o software seja “confiável”. Ninguém pediria para ser construído um software “não confiável”, porém diferentes clientes estão interessados em diferentes níveis de confiabilidade e diferentes formas de certificar esse nível.

Deve ser levado em conta que quanto mais esforço é colocado para se alcançar um requisito, maior é o custo agregado ao projeto e isso servirá como referência para o cliente escolher o grau de confiabilidade que deseja. Por isso um requisito deve ser verificável. O requisito “robusto”, por exemplo, pode ser medido por meio do **MTBF** sistema.

(Sommerville, 2015) mostra uma pequena taxonomia dos tipos de requisitos não-funcionais mais importantes, apresentada na Figura 8.4.

**MTBF** é o tempo médio entre falhas.

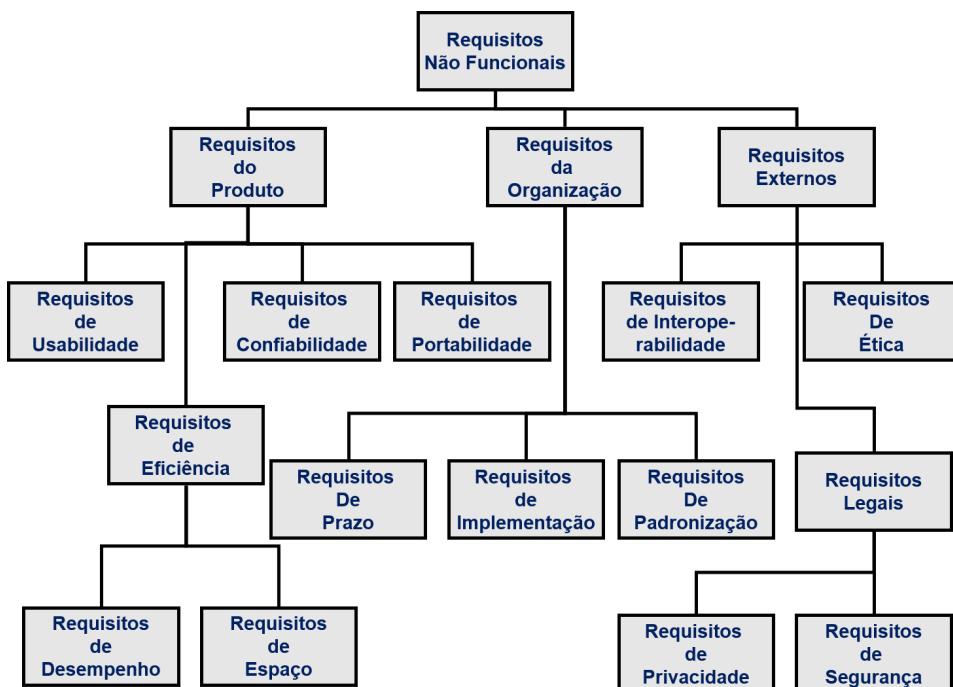


Figura 8.4.: Pequena taxonomia de requisitos não funcionais. Fonte: (Sommerville, 2015)

### 8.3.3. Restrições

É difícil a discussão para decidir se algo é um requisito ou uma restrição. Essa divisão é muitas vezes artificial e feita a partir de um ponto de vista específico. Por exemplo, se uma especificação obriga o sistema a ser desenvolvido em Java, isso é uma restrição ou um requisito?

Uma maneira de separar restrições de requisitos é analisar se a demanda vem da vontade dos clientes ou usuário diretos, ou é algo obrigatório para eles, como uma lei ou uma regra organizacional que limita as possibilidades do projeto. Ou seja, se o sistema deve ser feito em Java por uma obrigação qualquer, e existem soluções mais fáceis, talvez seja melhor considerá-lo uma restrição.

### 8.3.4. Outros tipos de requisitos

A literatura de requisitos é ampla e propõe muitas divisões para os requisitos, como requisitos de interface, requisitos de ambiente e etc. Essas divisões são úteis para organização de documentos de especificação de requisitos..

Requisitos de Informação representam a informação que o cliente deseja obter do sistema. Requisitos de informação também são atendidos por eventos. Muitas vezes o cliente expressa requisitos de informação de forma funcional. Outras vezes o cliente está preocupado em conseguir uma informação, mas não sabe como fazê-lo na forma de um requisito funcional. Em todo caso, sempre nos preocuparemos em levantar todos os requisitos de informação, pois eles representam as respostas fundamentais do sistema.

**Requisitos de interface** são exigências de como o sistema deverá interagir com outros sistemas ou humanos. Também são comuns na maioria das especificações formais. **Requisitos de desempenho** especifica requisitos numéricos estáticos e dinâmicos que são determinados para o sistema ou para a interação com o usuário(IEEE, 1998).

## 8.4. O Papel dos Requisitos no Projeto

J. Robertson e S. Robertson (2006) mostram o papel dos requisitos em um projeto de software de acordo com a Figura 8.5. O processo de Elicitação de Requisitos parte das Necessidades do Usuário e gera requisitos tanto para a Modelagem de Sistema (principalmente funcionais) quanto para o Projeto de Sistema (principalmente não funcionais). Ele também recebe feedback dessas e outras fases que podem levar a necessidade de correção dos requisitos. Modelos gerados pela Modelagem de Sistemas auxiliam na busca de novos requisitos, por exemplo, permitindo entender que informação está faltando para construir um modelo de dados correto.

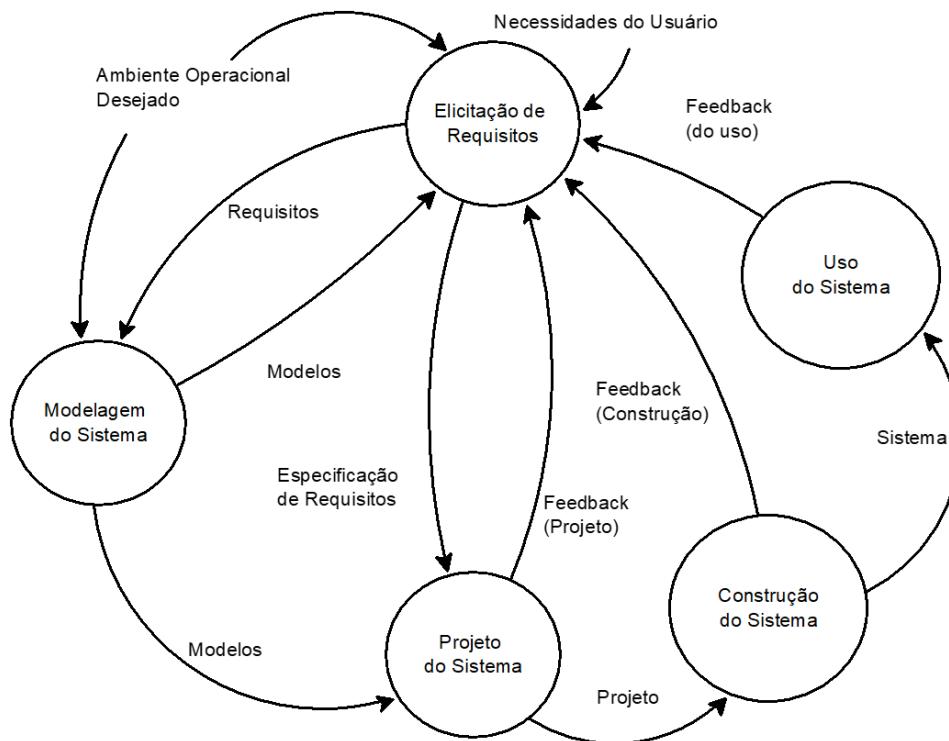


Figura 8.5.: O papel dos requisitos no desenvolvimento do sistema. Fonte:(J. Robertson e S. Robertson, 2006)

## 8.5. Engenharia de Requisitos

A Engenharia de Requisitos é a disciplina relacionada com o trabalho com requisitos, e pode ser dividida em duas grandes áreas: Desenvolvimento de Requisitos e Gerência de Requisitos, sendo que a última não é tratada neste texto. O Desenvolvimento de Requisitos, por sua vez, pode ser dividido em(K. E. Wiegers e Beatty, 2013):

- Elicitação, que cobre todas as atividades de descoberta dos requisitos;
- Análise, que cobre o entendimento mais rico e preciso de cada requisito;
- Especificação, que trata da representação e armazenamento dos conhecimentos coletados de forma persistente e bem organizada, e
- Validação, que trata da confirmação que os requisitos estão corretos e vão permitir uma solução que satisfaz as partes interessadas.

## 8.6. Elicitação de Requisitos

A **elicitação** é o levantamento, registro e validação (Christel e Kand, 1992) das expectativas dos diversos interessados no sistema, seguido da consolidação e validação

## 8. Requisitos

dessas expectativas em requisitos formais. Contemplar essas diferentes visões implica projetar interesses divergentes e conciliá-los.

Esse trabalho investigativo de descoberta dos requisitos, é feito com dois processos gerais: entrevistas ou reuniões, ou observação. A diferença básica é que no primeiro tipo de elicitação são feitas perguntas aos clientes, enquanto na segunda o analista observa o ambiente de trabalho da organização. O entendimento gerado pela processo de observação pode exigir entrevistas ou reuniões adicionais para confirmação dos achados. Existem muitas práticas específicas para ambos os processos, como entrevistas fechadas, entrevistas por meio de questionários, reuniões de vários formatos, observação direta, observação de cenas gravadas, análise de documentos, atuação no papel do usuário, etc. (Sommerville, 2015).

Já K. E. Wiegers e Beatty (2013) sugerem 8 técnicas de elicitação e indicam em que tipos de projetos são adequadas, como mostrado na Tabela 8.1.

Tabela 8.1.: Adequação das formas de elicitação propostas por K. E. Wiegers e Beatty (2013) e alguns tipos de projeto.

TIPOS DE PROJETO	TÉCNICAS DE ELICITAÇÃO					
	entrevistas	workshops	grupos de foco	observações	questionários	análise da interface do sistema
software para o mercado consumidor	✓		✓		✓	
software interno da organização	✓	✓	✓	✓		✓
troca de um sistema existente	✓	✓		✓		✓
melhoria de um sistema existente	✓	✓	✓		✓	✓
aplicação nova	✓	✓			✓	
pacote de software	✓	✓		✓	✓	✓
sistemas embarcados	✓	✓				✓
partes interessadas geograficamente distribuídas	✓	✓		✓		

Para levantar requisitos é necessária a interação entre aqueles que conhecem os requisitos ou a necessidades dos usuários e outras partes interessadas e os desenvolvedores. É um processo interativo de comunicação, onde, por aproximações sucessivas, o desenvolvedor constrói um modelo aceito pelos usuários, como ilustrado na Figura 8.6.

## *8.6. Elicitação de Requisitos*

O mesmo acontece na investigação que um analista tem que fazer com o cliente em busca do requisito. Inicialmente o cliente será ambíguo, generalista e paulatinamente, com a ajuda do analista, deverá chegar a uma especificação precisa do que deseja.

Uma receita geral para o levantamento de requisitos pode ser dada em 5 passos:

1. Identificar quem fornecerá os requisitos
2. Levantar lista de desejos para estas pessoas
3. Refinar cada lista de desejos até que seja auto-consistente
4. Integrar as listas de desejos de forma que sejam consistentes
5. Determinar os requisitos não funcionais.

Apesar de tudo, não é uma tarefa fácil levantar os requisitos. Muitos problemas podem acontecer. É comum que os clientes não saibam exatamente o que querem ou não saibam articular propriamente suas ideias, especialmente quando o desenvolvedor não possui o linguajar típico da área de aplicação (jargão). Muitas vezes, também, os desenvolvedores pensam entender melhor do problema que seus clientes, propondo supostas “melhorias” que afetam custo e aumento o risco dos sistemas propostos.

### **8.6.1. Processo de Elicitação de Requisitos**

Dentro do Processo de Desenvolvimento de Software existe um Processo de Elicitação de Requisitos. Christel e Kand (1992) sugeriu o processo descrito na Figura 8.7 e na Tabela 8.2:

## 8. Requisitos

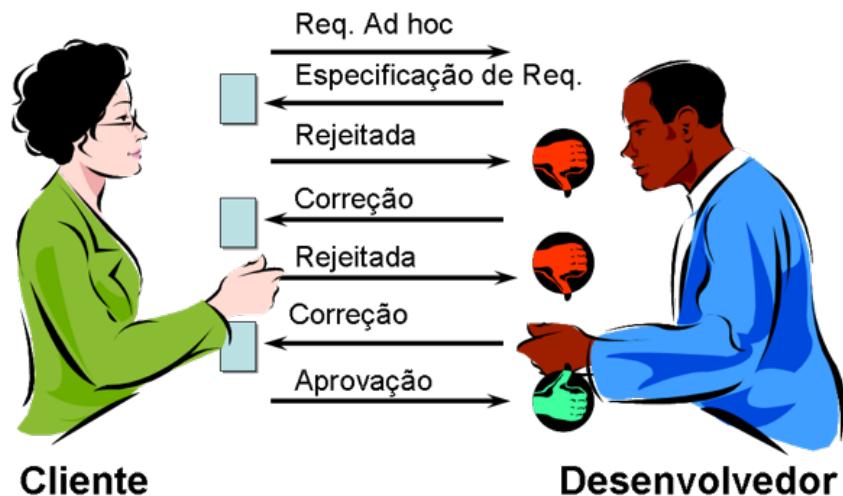


Figura 8.6.: A elicitação de requisitos é um processo interativo onde, por aproximações sucessivas, o desenvolvedor consegue a aprovação de uma especificação de requisitos.

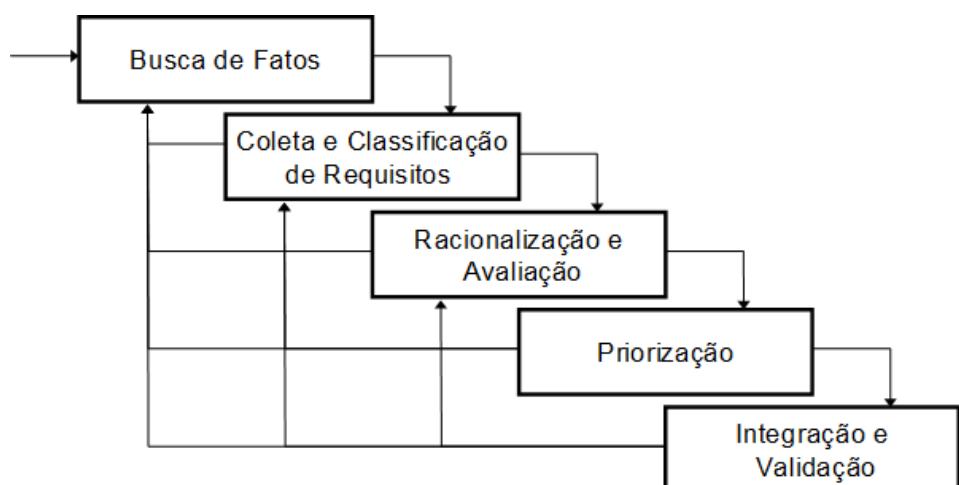


Figura 8.7.: O processo de elicitação de requisitos, adaptado de Christel e Kand (1992).

Tabela 8.2.: Processo de Elicitação de Requisitos segundo Christel e Kand (1992).

Tarefas da Elicitação de Requisitos		
Atividade	Tarefas orientadas ao usuário	Tarefas orientadas ao desenvolvedor
Busca de Fatos	<ul style="list-style-type: none"> <li>Identificar grupos relevantes através de múltiplos níveis da organização.</li> <li>Determinar os contextos operacional e do problema, incluindo a definição dos modos operacionais, objetivos e cenários de missão como apropriados.</li> <li>Identificar sistemas similares.</li> <li>Realizar análise de contexto.</li> </ul>	<ul style="list-style-type: none"> <li>Identificar especialistas do domínio da aplicação e de desenvolvimento.</li> <li>Identificar modelo de domínio e modelo de arquitetura.</li> <li>Conduzir pesquisas tecnológicas, para mais tarde fazer estudo de viabilidade e análise de risco.</li> <li>Identificar custos e restrições à implementação impostas pelo patrocinador.</li> </ul>
Coleta e Classificação dos Requisitos	Levantar a lista de desejos de cada grupo.	Classificar a lista de acordo com funcionais, não funcionais, restrições de ambiente, restrições de projeto e ainda de acordo com as partições definidas pelo modelo de domínio e pelo paradigma de desenvolvimento.
Racionalização e Avaliação	Responder questões da forma “Por que você precisa de X?”, a partir de raciocínio abstrato. Isso auxilia a transformar o raciocínio das questões sobre “como?” para as questões sobre “o quê?”.	Realizar uma análise de riscos, investigando técnicas, custos, prazos e incluindo análise de custos e benefícios e viabilidade baseado na disponibilidade da tecnologia.
Priorização	Determinar funcionalidades críticas para a missão.	Priorizar requisitos baseados em custo e dependência. Estudar como o sistema pode ser implementado de forma incremental, investigando modelos arquiteturais apropriados.
Integração e Validação	<ul style="list-style-type: none"> <li>Resolver a maior quantidade possível de pontos em aberto.</li> <li>Validar que os requisitos estão concordando com os objetivos iniciais.</li> <li>Obter autorização e verificação para passar ao próximo passo de desenvolvimento (e.g. a demonstração e a validação).</li> </ul>	Resolver conflitos e verificar consistência.

## 8.7. Níveis de Abstração dos Requisitos

Qualquer que seja o sistema, existem várias formas de entendê-lo. Similarmente, existem vários níveis de abstração de requisitos em que eles podem ser definidos, dependendo da visão necessária naquele instante. Alguns níveis de abstração reconhecidos para o início do projeto são(K. E. Wiegers e Beatty, 2013), na sequência em que normalmente aparecem:

1. **requisito do negócio** descrição de alto nível do objetivo do projeto que justifica a necessidade do projeto para o negócio, com visão dirigida para os objetivos do negócio;
2. **requisito do usuário** é algum comportamento ou característica que o usuário deseja do software ou o sistema como um todo; o que o usuário quer. São escritos pelo próprio usuário ou levantados por um analista de sistemas que consulta o usuário.
3. **requisito do sistema** é algum comportamento ou característica exigido do sistema como um todo, incluindo hardware e software. O comportamento desejado do sistema. São normalmente levantados por engenheiros ou analistas de sistemas, refinando os requisitos dos usuários e os transformando em termos de engenharia.
4. **requisito do software** é algum comportamento ou característica que é exigido do software. São normalmente levantados por analistas de sistemas.O objetivo da análise é capturar todos os requisitos para o software sendo desenvolvido e apenas aqueles requisitos verdadeiros.

## 8.8. O Risco de Requisitos Falsos

Uma especificação de software deve conter todos os requisitos e apenas requisitos verdadeiros(McMenamin e Palmer, 1984).

É óbvio que a falta de requisitos implica que o sistema não atenderá seus clientes, não atingindo o valor desejado. Já o requisito falso, ou seja, aquele que não é necessário para que o sistema cumpra sua finalidade(McMenamin e Palmer, 1984), implica em custos e riscos maiores ao projeto.

Segundo McMenamin e Palmer (1984) há dois tipos de requisitos falsos em uma especificação: os tecnológicos e os arbitrários.

Uma das formas do requisito falso tecnológico acontecer é pela transmissão de uma tecnologia de um sistema já existente para outro, quando ela não devia mais existir. Por exemplo, um sistema antigo que precisa avisar os usuários que algo aconteceu pode enviar avisos por *pager*, fax, SMS e correio eletrônico. Em uma nova versão, se adiciona o envio por *Whatsapp* e *Messenger*, mas não se remove o envio por *pager* e fax, tecnologia que não são mais usadas.

A segunda forma é pelo especificação antecipada de um requisito tecnológico. Requisitos tecnológicos falsos tendem a levar o sistema novo em uma direção de desenvolvimento que pode não ser a mais adequada(McMenamin e Palmer, 1984). Um exemplo seria em um projeto de longo prazo especificar já no início do projeto que deveria ser usado um *framework* específico de software, sem avaliar se o *framework* atende todas as capacidades exigidas. Ao longo do projeto, por exemplo, pode ser descoberto que é necessário que o software seja um web site responsivo, tecnologia que não necessariamente estaria coberta pelo *framework*.

Os requisitos falsos arbitrários aparecem quando o analista inclui um requisito por vontade própria ou por causa do uso de uma técnica de modelagem(McMenamin e Palmer, 1984).

Requisitos falsos tem dois efeitos no desenvolvimento de um projeto de software: eles podem esconder os requisitos verdadeiros e eles aumentam o tamanho do sistema, gerando custos e riscos maiores.

## 8.9. Características de um Bom Requisito

Um requisito, ou melhor, seu registro, deve ter as seguintes características (IEEE, 2018):

**necessário**, significando que, se retirado, haverá uma deficiência no sistema, isto é, ele não atenderá plenamente as expectativas do usuário;

- em especial, não devem existir requisitos do tipo “seria interessante ter”<sup>2</sup>, ou o requisito é necessário ou é dispensável;
- deve ser levado em conta que cada requisito aumenta a complexidade, o custo e o risco do projeto, logo, não podem ser introduzidos de forma espúria;

**apropriado**, a intenção e a quantidade de detalhe é adequada ao nível de abstração da especificação e a entidade a que se refere;

**não-ambíguo**, possuindo uma e apenas uma interpretação, sendo muito importante prestar atenção na linguagem sendo utilizada;

**completo**, ou seja, não precisa ser explicado ou aumentado, garantindo capacidade suficiente do sistema;

**consistente**, não contradizendo ou mesmo duplicando outro requisito e utilizando os termos da mesma forma que outros requisitos;

**singular**, ou seja, sendo apenas um requisito e não uma composição de requisitos;

**alcançável**, ou seja, realizável a um custo definido por uma ou mais partes interessadas no sistema ;

**rastreável**, de forma que sua fonte ou origem possa ser encontrada;

**verificável**, não sendo vago ou geral e sendo quantificado de uma maneira que permita a verificação na forma de inspeção, análise, demonstração ou teste, e

---

<sup>2</sup>Isso está em desacordo com a proposta de priorização da técnica MoSCoW

## 8. Requisitos

**correto**, sendo uma representação adequada, e **conforme**, de acordo com um padrão e estilo aprovados, quando aplicável.

Além desses fatores, quando um requisito for funcional, deverá ser também **independente da implementação**, ou seja, o requisito define o que deve ser feito, mas não como.

Um requisito mal elaborado pode ser transformado em um requisito por meio de seu detalhamento. Por exemplo, o requisito “O sistema deverá responder rapidamente” tem grande ambiguidade. O que é “rapidamente”? Em que contexto deve ser entendido? Esse requisito pode ser melhorado para “O tempo de resposta do sistema deverá ser de 2 segundos”. Esta forma, apesar de mais detalhada, gera uma dúvida: se o sistema for mais rápido do que isso, o tempo de resposta deve ser aumentado até chegar a 2 segundos? Queremos exatamente dois segundos? Podemos melhorar então para “O tempo de resposta do sistema deverá ser de no máximo de 2 segundos”. Já está bem melhor, mas isso se aplica a todas as partes do sistema? Alguns relatórios, por exemplo, não podem tomar mais tempo. O requisito é alcançável? Então podemos melhorar para dois requisitos “O sistema deverá buscar manter o tempo de resposta menor do que 2 segundos em todas suas funcionalidades” e “O sistema deverá avisar ao usuário de qualquer operação cujo tempo de resposta previsto for maior do que 2 segundos”.

### 8.9.1. Efeitos de Requisitos com Problemas

A importância de obter requisitos corretos pode ser compreendida rapidamente se imaginarmos que um requisito, quanto mais básico e importante ele é, mais partes do sistema vai afetar.

Basicamente, cada requisito do negócio vai gerar um ou mais requisitos de usuário, e cada requisito do usuário vai gerar um ou mais requisitos do sistema, que vão, por sua vez, gerar outros requisitos com mais detalhes, e isso vai se repetindo a cada fase do projeto de software. O efeito, com a evolução do projeto, é que cada requisito inicial representa não só um compromisso, mas um fator crítico em uma sequência de decisões que são tomadas ao longo do projeto e influenciam várias de suas partes.

Por exemplo, se um requisito for “O sistema deverá enviar uma mensagem de felicitações no dia do aniversário do usuário”, será necessário ter um campo com a data de aniversário no banco de dados, campos em telas que permitem entrar e corrigir a data de aniversário, uma função para mandar a mensagem, possivelmente uma tela, ou campo em uma tela existente, para cadastrar essa informação, etc. Várias partes do código então se referem a um só requisito.

Assim, os erros de requisito afetam drasticamente todo o projeto. Dados empíricos indicam que um problema nos requisitos, se detectado nas fases finais do projeto, pode custar 100 vezes mais para ser corrigido do que se for detectado na fase de elicitação de requisitos. Mesmo que essa detecção se faça mais cedo, em uma fase intermediária, esse custo ainda seria 10 vezes maior.

Um requisito que não caiba em uma das características de qualidade citadas será propenso a defeitos e aumentará o risco do projeto.

Se não for necessário, gastaremos esforços em sua definição e implementação que seriam mais bem gastos tratando de outros requisitos. Além disso, cada requisito adicional aumenta o risco do projeto como um todo, ainda mais se a adição for dispensável.

Se for ambíguo, corremos alto risco de implementá-lo errado, desagradando o cliente.

Se não for verificável, não poderemos testar de maneira inequívoca se está corretamente atendido, possibilitando discussões sobre sua realização.

Se não for completo, faltará ao projeto alguma coisa que o usuário necessita. Se não for consistente, o projeto chegará a encruzilhadas ou talvez se torne impossível.

## 8.10. Atributos dos Requisitos

A norma IEEE 29148 diz que requisitos bem formados devem ter incluir um conjunto de atributos descritivos que permitam entendê-los e gerenciá-los, e dá como exemplo(IEEE, 2018):

- identificação;
- versão;
- proprietário, isto é, quem mantém o atributo;
- prioridade para as partes interessadas;
- justificativa;
- risco;
- dificuldade, e
- tipo.

Veremos que a maioria desses atributos são normalmente informados nos métodos atuais, como histórias do usuário ou casos de uso, porém normalmente o risco<sup>3</sup> não é tratado requisito por requisito, sendo normalmente resultado de outra atividade de gerência de projeto ou análise de sistemas.

### 8.10.1. Volere e o *snowcard*

O Cartão Volere ou *snowcard* (J. Robertson e S. Robertson, 2006) é uma forma de registrar requisitos de software que atende parcialmente esta especificação, foi apresentado na Figura 8.1. O registro de requisitos em cartões possibilita que realização de algumas atividades interativas sejam realizadas com facilidade, como a ordenação por prioridade. O cartão apresentado na Figura 8.1, contém os seguintes campos:

---

<sup>3</sup>Lembramos que o risco aparece no Project Model Canvas, no capítulo ??

## 8. Requisitos

- **número identificador**, o para facilitar a discussão, identificamos todos os requisitos unicamente;
- **tipo**, classificando-o como funcional, não funcional, ou outra classificação mais detalhada usada na organização;
- evento, história do usuário, caso de uso, etc. que o atende;
- **descrição**, uma sentença que descreve o requisito;
- **justificativa**;
- **donde do requisito**, a pessoa, grupo, organização, documento ou outra que o origem para o requisito;
- **critério de aceitação**, uma medida que possa ser usada para garantir que o requisito foi alcançado;
- **satisfação do usuário**, um grau, de 1 (nenhum interesse) a 5 (extremamente satisfeito), por exemplo, indicando a satisfação do cliente se esse requisito for alcançado;
- **insatisfação do usuário**, um grau, de 1 (nenhum interesse) a 5 (extremamente insatisfeito), por exemplo, indicando a satisfação do cliente se esse requisito não for alcançado;
- **dependências**, referências a outros requisitos que dependem de alguma forma desse requisito;
- **conflitos**, referência aos requisitos que de alguma forma conflitam com esse
- **material de apoio**, listagem de material de apoio para atender esse requisito, e
- **histórico**, documentação da criação e das mudanças efetuadas.

Um diferencial do método Volere é que ele associa o valor a duas perguntas: a satisfação com a implementação do requisito e a insatisfação com sua ausência no produto. Isso permite classificar cada requisito em um de quatro quadrantes, de acordo com a nota que recebem: crítico, decoração, conflituoso ou dispensável (não fazer). A Figura 8.8 demonstra essa ideia.



Figura 8.8.: Cada requisito se posicionará em algum ponto desse gráfico, de acordo com as notas de avaliação.

## 8.11. Priorizando Requisitos

Existe uma tendência grande de o sistema crescer muito durante a análise. Principalmente se entrevistamos um grande número de pessoas, existe uma facilidade natural que elas têm para propor novas funcionalidades para um sistema que ainda não existe, por imaginarem alguma utilidade nessas funções propostas. Assim, muitas vezes nos vemos envolvidos com uma quantidade de requisitos tão grande que é óbvio que o sistema a ser feito não poderá ser entregue no prazo ou pelo custo combinado, ou que se pensava em combinar.

Nesse caso, algumas técnicas podem ser utilizadas para caracterizar o que deve ser realmente feito ou, pelo menos, em que ordem as coisas devem ser feitas. A priorização de requisitos é atividade essencial em qualquer projeto e natural em métodos ágeis que usam o *product backlog*.

A primeira técnica disponível é associar a cada requisito do sistema uma importância. Uma escala de três ou cinco valores é adequada para isso, como em: “Imprescindível para o sucesso do sistema”, “Funcionalidade Importante, mas podemos esperar algum tempo”, “Ajudaria ter, mas é possível viver sem essa funcionalidade”, “Benefícios mínimos”, “Desnecessário”.

A segunda técnica disponível é planejar o sistema para ser entregue em várias **versões**, mesmo que nem todas as versões estejam incluídas nesse contrato, e pedir para o cliente determinar que funcionalidades devem aparecer em cada versão. Nesse caso pode ser

## 8. Requisitos

interessante dar um peso ou custo para cada requisito, de modo que o cliente possa “controlar seus gastos”.

Uma terceira técnica disponível é dar uma **moeda virtual** para o cliente, por exemplo, 1000 dinheiros, e pedir para ele distribuir quanto pagaria por cada função, priorizando no desenvolvimento aquelas funções que o cliente decidir pagar mais.

Uma quarta técnica é conhecida como **MoSCoW**, onde cada item proposto é associado a um conceito do grupo “*Must Have*”, precisa ter, “*Should Have*”, devia ter, “*Could Have*”, podia ter e “*Won’t Have*”, não vai ter (IIBA, 2011), criando grupos com significados específicos;

A técnica **Volere** também separa os requisitos dentro de seu quadro de importância, como visto na Figura 8.8.

Todas essas técnicas, porém, ficam dependentes de uma outra priorização importante dos requisitos: a priorização por dependência.

Devem ser levados em conta os vários fatores que influenciam nessa determinação de prioridades, entre eles os citados por (Volare, 2020):

- Diminuir o custo da implementação
- Valor para o comprador
- Tempo para implementar
- Facilidade técnica de implementar
- Facilidade do negócio para implementar
- Valor para o negócio
- Obrigação por alguma autoridade externa

### 8.12. Requisitos Mudam com o Tempo

Também é importante perceber que os requisitos de um software mudam com o tempo. Essas mudanças ocorrem porque o ambiente em que o software reside também muda. Os países alteram suas leis, as organizações alteram suas práticas, a tecnologia evolui, os usuários exigem novas funcionalidades até então não imaginadas ou desnecessárias.

Caper Jones (2005), afirmou que a taxa de mudanças de requisitos para um software comercial em desenvolvimento chega a 3,5% ao mês, e para um software para Web, chega a 4,0% ao mês. Considerando o tempo médio de execução desses e outros tipos projetos, ele chega a valores médios de 2,58% de mudanças ao mês, 14 meses em média de execução e 32,33% de mudança total nos requisitos. É importante notar, porém, que todas essas afirmações são baseadas em cálculos com Pontos de Função para manutenção, onde pequenas alterações podem causar grandes efeitos. Além disso, modificações ao longo do projeto podem representar não uma mudança do requisito verdadeiro, mas sim uma compreensão melhor do que é esse requisito. Mesmo assim, uma mudança corretiva em um requisito tem o mesmo impacto de uma mudança por novas necessidades.

Esse fato tem efeitos em nossa prática. O primeiro é que devemos estar preparados para a mudança, pois ela vai acontecer. O segundo é que quanto maior o tempo de duração do projeto, maior a quantidade de mudanças de requisitos, o que aumenta ainda mais o risco, que já é afetado pelo projeto ser longo e provavelmente complexo.

Assim, não só é importante conhecer os requisitos, mas também conhecer qual a sua estabilidade. Requisitos pouco estáveis devem ser tratados de forma diferente dos requisitos mais estáveis. Requisitos mais críticos devem ser tratados de forma diferente dos requisitos opcionais. Isso não tem relação com o software apenas, mas sim com atender o negócio.

## 8.13. Requisitos e Necessidades

Requisitos são originários de necessidades das partes interessadas. Enquanto os requisitos vivem no “mundo das soluções”, as necessidades vivem no “mundo dos problemas”. Os requisitos indicam as características que devem ser observadas no sistema, que atendem as necessidades.

Voltando ao discutido no Capítulo 3, as necessidades causam desejos, que por sua vez causam demandas. Os desejos são influenciados pelo ambiente, as demandas pela capacidade financeira do comprador. Assim, se uma organização tem como necessidade tornar mais rápido seu processo de vendas, baixando de 15 minutos para 3 minutos o tempo de concluir uma venda, como estratégia de negócios ela pode desejar desde um aplicativo para celular para seus vendedores usarem na loja, como faz a Apple, ou uma loja toda automatizada onde o que você coloca no carrinho, ou tira do carrinho, é identificado por câmeras e posto, ou removido, automaticamente de sua conta, o que acontece no mercado da Amazon. Porém, o que ela pode pagar? Pode ser que apenas uma nova versão do seu software de caixa registradora, de forma que ele seja mais rápido. Os requisitos virão dessa demanda.

Como é visto no Capítulo 7, necessidades podem ser provenientes de problemas, e um problema é uma diferença, vista com uma perspectiva comportamental, entre uma situação percebida e uma situação desejada. No momento, nos basta a noção que o problema incomoda o usuário (ou a parte interessada) ao ponto dele considerar necessário investir alguma quantia ou esforço de forma a evitar que o problema aconteça.

É comum ouvirmos o ditado “Em time que está ganhando, não se mexe”. Quando somos chamados para desenvolver um sistema, devemos imaginar imediatamente que, se alguém deseja mexer em sua organização, então é porque “não está ganhando”, pelo menos da maneira que supõe possível. Faz pouco sentido imaginar que alguém iria fazer um investimento de risco, e sempre há risco no desenvolvimento de software, sem que haja uma necessidade clara. Se isso for identificado, talvez seja mais adequado cancelar o projeto. Não se deve esquecer o adágio contrário: “em time que está vencendo que se mexe”. Essa frase representa a ideia que para continuar “ganhando” é necessário estar em

## 8. Requisitos

constante evolução, e isso pode ser também uma motivação importante para um projeto de software.

Muitas vezes o analista se depara com a necessidade de, antes de definir requisitos, definir propriamente qual o problema a ser tratado. Para isso, deve fazer com que os clientes cheguem a um acordo sobre qual é o verdadeiro problema, ou o problema mais importante. Faz pouco sentido construir um sistema se o verdadeiro problema de negócios não for endereçado pela solução planejada. Isso é tratado no Capítulo 7

Um sistema pequeno certamente produzirá a solução para uma pequena quantidade de problemas, um sistema grande certamente atingirá uma grande quantidade de problemas. O ponto em questão é que deve existir alguma não conformidade, seja ela atual ou futura, para valer a pena o investimento, e o risco, de tentar um sistema novo.

### 8.14. Requisitos de Acordo com Normas IEEE

A IEEE tem um conjunto de normas que tratam de requisitos. Esta seção trata de alguns dos assuntos levantados por essas normas.

Segundo a norma IEEE 24765:2010 (IEEE, 2018) requisitos devem dizer o que (*what*) é desejado, e não como (*how*), sem incluir nenhuma decisão de projeto nos seus níveis mais abstratos.

A linguagem da especificação de requisitos deve evitar termos gerais e abstratos, ou que geram requisitos difíceis de verificar. Devem ser evitados(IEEE, 2018):

- adjetivos e superlativos, como “bom” ou “melhor”;
- linguagem subjetiva, como “amigável ao usuário” ou “custo eficiente”;
- pronomes vagos, como “aquele”;
- termos ambíguos como advérbios e adjetivos, como “quase sempre” ou “mínimo”;
- termos lógicos que em português podem ser confusos, como “e” ou “ou”;
- termos não verificáveis, como “fornece suporte”;
- frases comparativas ambíguas, como “melhor que” ou “de melhor qualidade”;
- termos que não são um compromisso definitivo, como “se possível”
- termos que implicam totalidade, como “nunca” ou “sempre”, pois são difíceis ou impossíveis de verificar, e
- referências incompletas, como falta de uma versão ao especificar um software a ser usado.

Ainda segundo a norma IEEE 29148 (IEEE, 2018) é comum estipular o significado uso de palavras que definem obrigatoriedade, futuro ou proibição<sup>4</sup>. A norma ainda diz que requisitos ágeis, como histórias do usuário, não precisam seguir essa regra.

A norma IEEE 29148 (IEEE, 2018) apresenta uma sintaxe para os requisitos com a forma:

---

<sup>4</sup>Em inglês são usadas palavras como “shall”, “must”, “should” e “shall not”.

[condição][sujeito][ação][objeto][restrição]

onde:

- **condição** é uma regra para ativação do requisito, seja ela o acontecimento de um evento ou um fato que se torna realidade;
- **sujeito** descreve um ator, normalmente o próprio sistema sendo especificado;
- **ação** descreve a ação principal do requisito;
- **objeto** descreve o objeto da ação, e
- **restrição** limita a forma como a ação funciona.

As sintaxes de exemplo apresentadas para requisitos na norma IEEE 29148 são(IEEE, 2018) são:

- [sujeito] [ação] [objeto] [restrição a ação], como em “O sistema acadêmico [sujeito] deve calcular [ação] a aprovação de cada aluno [objeto] segundo as regras do curso específico [restrição]”, ou
- [condição] [sujeito] [ação] [objeto] [restrição a ação], como em “Ao fim do período [condição] o sistema acadêmico [sujeito] deve calcular [ação] a aprovação de cada aluno [objeto] segundo as regras do curso específico [restrição]”.

### 8.14.1. Requisitos Funcionais na Norma IEEE

Segundo a norma IEEE 830:1998 (IEEE, 1998) os requisitos funcionais devem “definir as ações fundamentais que devem acontecer no software ao aceitar e processar as entradas e no processamento e geração das saídas”. Normalmente, devem ser listados com uma sentença que se inicia com “O sistema deve”<sup>5</sup>. No contexto dessa norma, os requisitos funcionais incluem(IEEE, 1998):

- validações de entradas;
- sequência exata de operações;
- respostas a situações anormais, como sobrecarga, falhas de comunicação e tratamento e recuperação de erros;
- efeitos dos parâmetros, e
- relacionamentos das saídas com as entradas, incluindo sequências de entradas e saídas e fórmulas de conversão de entradas para saídas.

Exemplos de requisitos funcionais seguindo essa norma são:

- o sistema acadêmico deve permitir que um aluno solicite seu boletim;
- o sistema acadêmico deve permitir que um professor dê as notas dos alunos para as turmas em que está alocado, e
- o sistema acadêmico deve permitir a um coordenador incluir um aluno em uma turma apesar de ser contra as regras de inclusão.

---

<sup>5</sup>Em inglês, “The system shall”

## 8. Requisitos

Para evitar confusões entre um sistema funcionando hoje em dia e um sistema sendo especificado para o futuro, é possível colocar o verbo deve no futuro, ficando o início do requisito “O sistema deverá”.

### 8.14.2. O Documento de Especificação de Requisitos

Uma especificação de requisitos de sistema é um documento que reúne os requisitos definidos e aceitos por todos os interessados naquele sistema.

Os assuntos que devem ser tratados em uma especificação de requisitos são, segundo a norma IEEE ISO/IEC/IEEE International Standard - *Systems and software engineering – Life cycle processes – Requirements engineering* (IEEE, 1998) são:

- funcionalidade, ou o que o software tem que fazer?
- interfaces externas, com quem ou com que sistemas o software interage?
- desempenho, qual a velocidade, disponibilidade, tempo de resposta das várias funções, etc.?
- atributos (de qualidade), quais são as considerações de portabilidade, correção, manutenibilidade, segurança, etc.?
- restrições de projeto impostas a implementação, como padrões a serem seguidos, políticas de integridade da base de dados, limites dos recursos disponíveis, ambiente de operação, etc.?

Segundo o mesmo padrão, um sumário adequado a uma especificação de requisitos seria(IEEE, 2018):

1. Introdução
  - a) Objetivo do Sistema
  - b) Escopo do Sistema
  - c) Visão Geral do Sistema
    - i. Contexto do Sistema
    - ii. Funções do Sistemas
    - iii. Características dos Usuários
  - d) Definições
2. Referencias
3. Requisitos do Sistema
  - a) Requisitos Funcionais
  - b) Requisitos de Usabilidade
  - c) Requisitos de Desempenho
  - d) Requisitos de Interface
    - i. Requisitos de Interface Externa
    - ii. Requisitos de Interface Interna
  - e) Operações do Sistema
  - f) Modos e Estados do Sistema
  - g) Características Físicas

- h) Condições Ambientais
  - i) Requisitos de Segurança
  - j) Requisitos de Gerência de Informação
  - k) Requisitos Regulatórios e Legais
  - l) Requisitos de Sustentação do Ciclo de Vida do Sistema
  - m) Requisitos de Empacotamento, Trato, Envio e Transporte
4. Verificação (paralelo as subseções da seção 3)
  5. Apêndices
    - a) Suposições e Dependências
    - b) Acrônimos e Abreviações

## 8.15. A Visão do Novo Sistema

Dar uma visão do sistema sendo proposto é uma maneira de definir requisitos de altíssimo nível com facilidade de entendimento para as partes interessadas.

A Visão do Sistemas é uma descrição de como vai funcionar o novo sistema em uma descrição simples, em linguagem corrente. Essa visão deve ser escrita com forte apoio do cliente, senão pelo próprio cliente.

Possivelmente, essa visão é apresentada junto com uma visão do sistema atual, com efeito comparativo, sendo ambas no mesmo nível de abstração e dentro de um mesmo documento inicial do projeto.

A visão pode incluir o protótipo de algumas telas do novo sistema, com a finalidade de mostrar a diferença do sistema novo para o velho ou ainda mostrar como será o comportamento de uma nova finalidade.

A visão do sistema pode incluir não só o funcionamento do sistema, mas também expectativas de comportamento e de efeitos do sistema no negócio. Deve ficar claro que a visão do sistema é uma declaração do usuário, não necessariamente um comprometimento do desenvolvedor.

A visão do sistema inclui também os requisitos já detectados, informalmente, pelo analista. Esses requisitos podem ser divididos em requisitos funcionais, requisitos de informação e requisitos não funcionais. Obviamente só estamos interessados em requisitos verdadeiros.

## 8.16. Descrevendo Requisitos

Normalmente as especificações de requisitos são escritas em linguagem natural (inglês ou português, por exemplo). O problema é que a forma como falamos e normalmente escrevemos é bastante ambígua. Isso exige que adotemos algumas técnicas básicas,

## 8. Requisitos

principalmente um formato padronizado, um estilo de linguagem e uma organização que facilite a manipulação do conjunto de requisitos.

Algumas dicas para escrever requisitos são (K. Wiegers, 1999):

- use sentenças e parágrafos curtos;
- use a voz ativa;
- use verbos no futuro;
- use os termos de forma consistente e mantenha um glossário;
- para cada requisito, avalie se a partir de sua definição é possível determinar se ele está pronto ou não;
- garanta que todos os requisitos são verificáveis imaginando (e possivelmente documentando) uma forma de fazê-lo;
- verifique requisitos agregados (termos como “e” e “ou” são uma boa indicação) e divida-os, e
- mantenha um nível de detalhe único em todos os requisitos.

### 8.16.1. Exemplos de Requisitos

A seguir listamos alguns exemplos de requisitos bem descritos, seguindo o estilo de exemplos originais de Karl Wiegers:

- O sistema deverá permitir que um paciente marque uma consulta.
- O sistema deverá confirmar que a consulta foi aceita pelo paciente.
- O sistema deverá permitir que um condômino solicite a segunda via de sua conta condominial.
- O sistema deverá permitir um aviso ao condômino que não pagar sua no prazo correto.
- O sistema deverá permitir que um fornecedor cadastre um produto no catálogo.
- O sistema deverá informar ao sistema de estoques que um produto foi vendido.

Todas as sentenças acima usam, na verdade, uma expressão do tipo “deverá”, tradução do inglês “shall”, que é uma forma tradicional do americano definir uma ordem. Talvez as seguintes descrições sejam mais adequadas a nossa língua:

- O sistema permitirá que um paciente marque uma consulta.
- O sistema confirmará ao paciente que a consulta foi marcada.
- O sistema permitirá que um condômino solicite a segunda via de sua conta condominial.
- O sistema permitirá um aviso ao condômino que não pagar sua no prazo correto.
- O sistema permitirá que um fornecedor cadastre um produto no catálogo.
- O sistema informará ao sistema de estoques que um produto foi vendido.

Também devemos notar que as entradas do usuário são descritas como “deverá permitir”. Alguns autores, como o próprio Wiegers, admitem que uma especificação de requisito exija algo do usuário, como em:

- O paciente deverá especificar qual a especialidade médica que deseja.

Essa abordagem não é adequada, pois está fazendo uma exigência ao usuário e não ao sistema. Uma abordagem mais apropriada seria:

- O sistema exigirá que o usuário especifique a especialidade médica que deseja.

Desta forma deixamos claro que:

1. O requisito é do sistema
2. Cabe ao sistema exigir do usuário a resposta
3. Não cabe ao usuário saber o que fazer, mas sim ao sistema saber o que o usuário tem que fazer.
4. A especificação de requisitos do software não requer nada ao usuário.

Em conclusão, uma especificação de requisitos só deve exigir funcionalidade do sistema sendo definido. Entendemos que muitas vezes um sistema faz exigências ao ambiente. Essas exigências podem algo como “o usuário deve falar inglês” ou “o computador deve ser do compatível com chips XYZ/2001”. Todas as exigências que o sistema faz devem ser documentadas. Uma localização específica do documento para isso é a seção de suposições ou dependências, mas demandas que o sistema faz não podem ser vistas realmente como um requisito do sistema, apesar de possivelmente definir características do sistema.

## 8.17. Exercício

### **Exercício 8.1:**

Escreva um conjunto de requisitos funcionais na forma de “O sistema deve...” para um aplicativo de celular que sirva como relógio e despertador, permitindo vários alarmes.

### **Exercício 8.2:**

Procure na internet os seguintes documentos:

- *Software Requirements Specification for Cafeteria Ordering System prepared by Karl Wiegers v1.0 de 2002*, e
- *Cafeteria Ordering System Vision and Scope Document prepared by Karl Wiegers v1.0 de 2002*.

Preste atenção que o primeiro documento tem duas versões na rede, uma com casos de uso, outra com especificações em linguagem natural. Baixe os dois. Analise os três documentos e guarde-os para exercícios futuros.

### **Exercício 8.3:**

Baseado no texto *Cafeteria Ordering System Vision and Scope Document prepared by Karl Wiegers v1.0 de 2002*, feito com linguagem Natural, escreva um documento semelhante para o sistema descrito no Capítulo ??

## 8. Requisitos

### **Exercício 8.4:**

Baseado no texto *Software Requirements Specification for Cafeteria Ordering System prepared by Karl Wiegers v1.0 de 2002*, feito com linguagem Natural, escreva um documento semelhante para o sistema descrito no Capítulo ??

### **Exercício 8.5:**

Vá para o site <http://jogodeanalisedesistemas.xexeo.net/> e visite a Livraria Resolve. A partir da sua visita especifique no formato IEEE todos os requisitos que você encontrar.

### **Exercício 8.6:**

Identifique todos os problemas na sua lista de requisitos para a Livraria Resolve. Defina perguntas no formato 5W2H para resolver esses problemas.

# **Parte III.**

## **Gestão de Software**



# 9

## O Que é um Projeto

Let our advance worrying  
become advance thinking and  
planning.

(Winston Churchill)

### Conteúdo

---

9.1.	Recursos e Entregas . . . . .	152
9.2.	Partes Interessadas . . . . .	153
9.3.	Sucesso, Fracasso e Riscos . . . . .	153
9.4.	Pontos Críticos ou Pontos Chave de Sucesso . . . . .	154
9.5.	Conclusão . . . . .	155

Este capítulo faz uma introdução ao que é um projeto.

Segundo o PMI (2017), um **projeto** é “um esforço temporário empreendido para criar um produto, serviço ou resultado exclusivo. Os projetos e as operações diferem, principalmente, no fato que os projetos são temporários e exclusivos, enquanto as operações são contínuas e repetitivas.”

Exemplos de projeto, de complexidades diversas, são:

- construir uma casa;
- elaborar um plano de marketing;
- consertar um equipamento;
- fazer uma inspeção de segurança;

## 9. O Que é um Projeto

- preparar um curso de treinamento, e
- desenvolver um software.

Os projetos tem como características(Kerzner, 2017, pg.2):

- ter um **objetivo específico**, cujo escopo pode ser elaborado tanto no início quanto, mais modernamente, ao longo do seu ciclo de vida(PMI, 2017, pg. 13);
- possuir um **prazo**, no qual deve ser completado dentro de certas especificações;
- ter **início e fim** definidos;
- ter uma **limitação de recursos**, e verba, quando aplicável,
- utilizar recursos, humanos ou não, e
- ser multifuncionais.

### 9.1. Recursos e Entregas

Projetos entregam alguma coisa aos seus clientes. Existem entregas físicas, como mesas, estradas, protótipos e equipamentos, e entregas abstratas, como software, relatórios e estudos, ou mesmo o resultado de um treinamento, como o aprendizado de um tema como “programação em C” por algumas pessoas. Um projeto sempre tem pelo menos uma entrega, que é o seu objetivo final, mas normalmente existem também entregas intermediárias, principalmente quando há um contrato , e a parte contratada precisa fornecer essas **entregas intermediárias** para comprovar que está fazendo sua parte e receber um pagamento, previamente acordado, da parte contratante.

Para poder fazer suas entregas, um projeto precisa utilizar vários tipos de recursos. Os tipos mais comuns são(Kerzner, 2017, pgs. 11, 12):

- dinheiro;
- pessoas;
- equipamentos e ferramentas;
- locais;
- infraestrutura;
- materiais consumíveis, e
- informação e tecnologia.

Outro item que poderia ser chamado de recurso é o tempo. Um projeto ocorre em um espaço de tempo, partindo de uma data de início e possuindo uma data de fim estimado. Atrasos no fim do projeto afetam negativamente o entendimento do seu sucesso, porém acabar um projeto antes do previsto nem sempre é considerado um sucesso adicional. Pela necessidade de gastar dinheiro continuamente durante um projeto, para garantir sua execução, um atraso normalmente aumenta o custo total do projeto.

## 9.2. Partes Interessadas

Todo projeto possui **partes interessadas**, que são indivíduos ou organizações que afetam ou são afetadas pelo projeto(PMI, 2017), seja favorável ou desfavoravelmente(PMI, 2017). Esses efeitos podem surgir ao longo, como o barulho causado em uma obra na rua, ou só ao fim do projeto, como a poluição que pode ser causada quando uma usina térmica de energia começa a funcionar. Vantagens e desvantagens geradas por um projeto podem causar conflitos de interesse entre as partes interessadas. Partes interessadas são tratadas detalhadamente no Capítulo 5.

## 9.3. Sucesso, Fracasso e Riscos

O sucesso de um projeto acontece quando ele se completa(Kerzner, 2017, pg. 6):

- dentro do prazo;
- dentro do orçamento;
- com o desempenho apropriado ou no nível especificado;
- sendo aceito pelo clientes e usuários;
- satisfazendo as partes interessadas (PMI, 2017, pg. 552)
- **com mudanças mínimas de escopo acordadas entre as partes interessadas;**
- sem causar distúrbio ao fluxo de trabalho da organização, e
- quando desejado, sem mudar a cultura da organização<sup>1</sup>.

As mudanças de escopo são normais em projetosKerzner (2017, pg. 6), provocadas pelo acontecimento dos riscos ou por mudanças não previstas no ambiente ou nas demandas das partes interessadas. Quanto mais longo o projeto, maior o risco de mudanças importantes. Mudanças, porém, podem gerar riscos adicionais, como afetar a moral da equipe. Em projetos que seguem a linha prescritiva de gerência, ou são mantidos sobre contratos fortes, as mudanças são mantidas em um mínimo necessário e exigem a concordância não só do cliente como da equipe de desenvolvimentoKerzner (2017).

### 9.3.1. A Incerteza dos Projetos e os Riscos

Todo projeto está associado a um nível de incerteza de seus resultados e entregas (Satpathy et al., 2016), não sendo possível garantir com absoluta certeza que um projeto terminará dentro de todos os parâmetros combinados. As técnicas de gerência de projeto tem como objetivo diminuir essas incertezas, principalmente com o uso de técnicas de análise de risco e do uso de estimativas e margens de erro.

Uma questão importante de projetos são seus riscos, ou seja, os desafios que podem ocorrer ao longo do mesmo e que temos de evitar ou mitigar para completar o projeto

---

<sup>1</sup>alguns projetos tem o objetivo de mudar a cultura

## *9. O Que é um Projeto*

com sucesso. Riscos acontecem com uma probabilidade, gerando um custo adicional, se acontecerem, e impactando o projeto de alguma forma, até mesmo com seu cancelamento. Um risco típico de projetos de software é perder pessoal.

Para garantir o sucesso do projeto é preciso gerenciá-lo ativamente. Existem várias formas de gerenciar projetos, e elas incluem atividades para iniciar, planejar, executar, monitorar e controlar e fechar projetos. O PMBOK (PMI, 2017) é uma fonte de referência importante que consolida várias técnicas tradicionais de gerência de projetos, como documentos necessários, diagramas de rede, uso da estrutura analítica de projeto e outras. Ele possui uma parte adicional para métodos ágeis e outra para projetos de software.

### **9.3.2. Benefícios da Gerência de Projeto**

Os benefícios potenciais de gerenciar um projeto são Kerzner (2017, pg. 3):

- identificação clara das responsabilidades funcionais para garantir que todas as atividades estão com responsáveis, mesmo com a troca de profissionais;
- minimizar a necessidade de relatos contínuos;
- identificar os limites de tempo para todos os agendamentos;
- identificar a metodologia para análise de custo-benefício;
- medir o que foi alcançado em relação ao plano;
- identificação antecipada de problemas para permitir ações corretivas;
- melhor capacidade de estimativa e planejamento, e
- conhecimento dos objetivos que podem não ser alcançados ou não ser excedidos.

## **9.4. Pontos Críticos ou Pontos Chave de Sucesso**

Os pontos críticos (ou chave) de sucesso para um projeto são aquelas questões que, não estando diretamente relacionadas ao desenvolvimento propriamente dito, são essenciais para o bom andamento do projeto.

Exemplos: compromisso de certos funcionários, fornecimento de certa informação, chegada de alguma máquina ou software, compromisso de entrega de dados ou equipamentos pelo cliente, etc.

Os pontos críticos devem ser levantados detalhadamente, pois os compromissos do desenvolvedor só poderão ser cumpridos caso sejam resolvidos satisfatoriamente.

## **9.5. Conclusão**

O desenvolvimento de um software é um projeto, e como tal deve ser gerenciado para aumentar as chances que tenha sucesso. Além disso, desenvolver software é um tipo especial de projeto que tem se mostrado, desde o início, muito desafiador, inclusive criando uma nova área de trabalho, a Engenharia de Software.

A análise de sistemas é uma das tarefas existentes em um projeto de software, e, dentro de todas as outras, tem como finalidade definir os requisitos verdadeiros desse projeto.



# 10

## Gestão de Projetos de Software

Plans are worthless. Planning is everything.

(Dwight D. Eisenhower )

### Conteúdo

---

10.1.	Técnicas Tradicionais de Gestão de Projeto . . . . .	158
10.2.	Uso das Técnicas Tradicionais . . . . .	166
10.3.	Técnicas Ágeis de Gestão de Projeto . . . . .	166

#### Por que Gestão de Projetos de Software?

To be done

Projetos podem ser divididos em quatro grandes momentos: iniciação, planejamento, execução e fechamento. Eles podem acontecer em sequência e em alguma forma cíclica. Esses momentos também podem ser únicos ou se repetir em sequência, de acordo com fases, que podem também ocorrer em paralelo.

No desenvolvimento de software essas fases e forma de sequenciamento ou paralelismo são definidas de acordo com um Processo de Desenvolvimento de Software.

O PMI (2017) define um processo de gerenciamento de projeto com cinco grupo de atividades: Iniciação, Planejamento, Execução, Encerramento, e Monitoramento e Controle. Para cada grupo define um conjunto de processos. Alguns desses processos são tratados ao longo deste livro. Por exemplo, no grupo de Iniciação se encontra o

## 10. Gestão de Projetos de Software

processo de “Identificar as Partes Interessadas”, tratadas no Capítulo 5. Já no grupo de Planejamento se encontram processos como “Coletar os Requisitos” e “Definir o escopo”, que também são tratadas em outros capítulos.

Nesse capítulo são discutidas algumas ferramentas que servem para o planejamento e depois continuam a ser usadas para a monitoramento e controle. A ideia aqui é tratar das questões ligadas a entrega de resultados do projeto no tempo esperado.

### 10.1. Técnicas Tradicionais de Gestão de Projeto

Entre as técnicas mais tradicionais para o planejamento existem três que são destinadas, simultaneamente ao planejamento e ao acompanhamento do projeto: PERT/CPM, Controlgrama de Gantt e WBS/EAP.

Essas técnicas apresentam como maior vantagem a facilidade de uso, porém elas são desafiadas pela singularidade de cada projeto de software e pela linearidade com que tratam o desenvolvimento de projetos. Na verdade, elas são muito adequadas para a construção de navios e prédios, mas apresentam problemas quando usadas no desenvolvimento de software.

É possível perguntar por que elas são ainda estudadas e usadas? A verdade é que elas fornecem uma boa visão do que poderia acontecer se “tudo desse certo” e os riscos que são corridos caso algo dê errado.

Hoje em dia quase toda técnica ou ferramenta ágil apresenta uma versão do Controlgrama de Gantt ou do WBS/EAP. Já as redes PERT/CPM são menos usadas, mas ainda são úteis para planejamento de projetos que não contam só com software, ou para planejamento a prazo mais longo. Além disso todas essas técnicas acabam, muitas vezes, sendo exigências contratuais.

#### 10.1.1. Estrutura Analítica de Projeto

Uma **Estrutura Analítica de Projeto (EAP)**, ou **Work Breakdown Structure (WBS)** é uma subdivisão das entregas ou do trabalho do projeto em componentes menores e mais facilmente gerenciáveis (PMI, 2017). O resultado é um visão estruturada do que será entregue.

Ele é estruturada na forma de um árvore exaustiva, hierárquica, do mais geral para o mais específico, e orientada por (PMI, 2017):

- entregas;
- fases de ciclo de vida, ou
- sub-projetos

O objetivo de uma EAP é identificar os elementos terminais da hierarquia, ou seja, os produtos, serviços e resultados a serem entregues pelo projeto (PMI, 2017).

Na sua representação, o nível superior contém apenas um nó que representa todo o projeto.

No nível mais baixo estão os **pacotes de trabalho**, que podem ser usados para agrupar atividades onde o trabalho é planejado, estimado, monitorado e controlado. Nesse caso, trabalho se refere aos produtos ou entregáveis que são o resultado da atividade, não a atividade (PMI, 2017).

Os pacotes de trabalho, como itens terminais, não são subdivididos. Eles devem poder ser estimáveis em termo de recursos, orçamento e duração necessários, ligados por dependência e agendados. Caso um candidato a pacote de trabalho não possa ser estimado, deve ser subdividido até que isso seja possível.

Os níveis intermediários podem ser organizados de várias formas diferentes:

- fases do projeto
- entregas principais
- subcomponentes

A EAP deve incluir 100% do trabalho definido pelo escopo do projeto e capturar todas as entregas — internas, externas ou intermediárias — de forma ao trabalho estar completo, incluídas as tarefas de gerenciamento de projeto. Isso vale para todos os níveis da hierarquia, isto é, a cada nível, o projeto deve estar descrito de forma completa em relação aquele nível. Obviamente, nenhum trabalho fora do escopo do projeto pode estar na EAP.

A EAP é uma técnica tradicional no planejamento do projeto e é necessário por indicar que entregas são desejadas, apesar de não ainda discutir o momento que são desejadas, mas possivelmente apenas sua ordem. Hoje em dia há formas similares a EAP usadas em projetos ágeis, como *Story Maps* (Patton e Economy, 2014b).

A lista numerada da Figura 10.1 representa uma possível EAP para um projeto de software que segue um processo simples em cascata. A mesma EAP está representada de forma gráfica na Figura 10.2.

### 10.1.2. Controlgrama de Gantt

Os **Controlgramas de Gantt**, desenvolvidos por Henry L. Gantt no início do século XX, são representações gráficas utilizadas para o planejamento e gestão de projetos. O diagrama ilustra visualmente a linha do tempo de um projeto, representando cada tarefa como uma barra horizontal, cujo comprimento corresponde à duração da tarefa. Isso permite que gestores de projeto e membros da equipe visualizem e monitorem facilmente o progresso das tarefas ao longo do tempo. Um exemplo dessa visualização é dado na Figura 10.3.

O Controlgrama de Gantt, como visto na Figura 10.3 também mostrar as relações entre as tarefas, isso é dependências, nas quais uma tarefa não pode começar até que

## 10. Gestão de Projetos de Software

1. Software de Agenda Telefônica
  - 1.1. Especificação
    - 1.1.1. Especificação da tela de inclusão
    - 1.1.2. Especificação da tela de edição
    - 1.1.3. Especificação da tela de exclusão
    - 1.1.4. Especificação da tela de listagem
    - 1.1.5. Especificação da tela de busca
  - 1.2. Projeto
    - 1.2.1. Projeto da tela de inclusão
    - 1.2.2. Projeto da tela de edição
    - 1.2.3. Projeto da tela de exclusão
    - 1.2.4. Projeto da tela de listagem
    - 1.2.5. Projeto da tela de busca
  - 1.3. Programação
    - 1.3.1. Programação da tela de inclusão
    - 1.3.2. Programação da tela de edição
    - 1.3.3. Programação da tela de exclusão
    - 1.3.4. Programação da tela de listagem
    - 1.3.5. Programação da tela de busca
  - 1.4. Testes
    - 1.4.1. Testes da tela de inclusão
    - 1.4.2. Testes da tela de edição
    - 1.4.3. Testes da tela de exclusão
    - 1.4.4. Testes da tela de listagem
    - 1.4.5. Testes da tela de busca

Figura 10.1.: EAP em formato texto, com o mesmo significado da Figura 10.2.

Fonte: do autor

outra seja concluída. Elas são geralmente indicadas com setas conectando as barras relevantes. Essa relação, porém, é mais clara nos Diagramas de Rede.

Embora os Controlgramas de Gantt sejam úteis para retratar a linha do tempo eles têm limitações. Por exemplo, podem não representar facilmente a complexidade de projetos grandes ou os recursos alocados para cada tarefa. No entanto, softwares modernos de gestão de projetos frequentemente combinam a visualização do Diagrama de Gantt com ferramentas e funcionalidades adicionais para tratar destas preocupações.

### 10.1.3. Redes PERT/CPM

O *Program Evaluation and Review Technique (PERT)* e o *Critical Path Method (CPM)* são técnicas que têm como objetivo auxiliar no planejamento e controle de projetos. Essas técnicas são frequentemente empregadas na gestão de projetos para

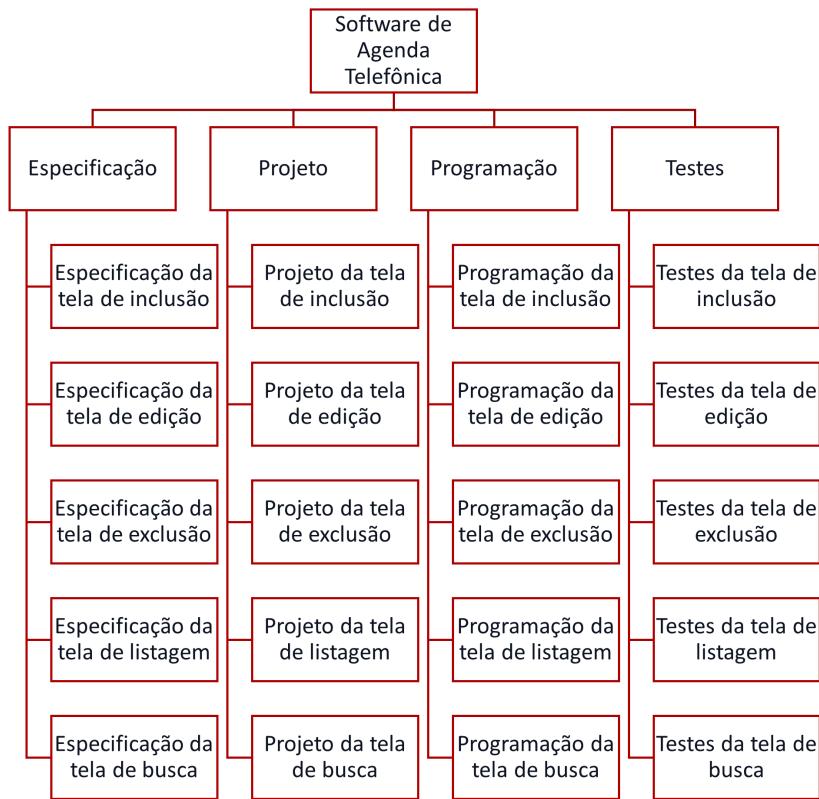


Figura 10.2.: Diagrama com modelo EAP equivalente ao da Figura 10.1. Fonte: do autor

determinar a duração total do projeto e identificar o caminho crítico, que é a sequência de tarefas que não pode sofrer atrasos sem afetar a duração total do projeto.

O PERT foi originalmente desenvolvido durante os anos 1950 pela Marinha dos EUA, com contribuições da Lockheed Martin e da Booz Allen Hamilton, para o programa de mísseis balísticos Polaris (Malcolm et al., 1959). Paralelamente, o CPM foi desenvolvido por Morgan R. Walker da DuPont e James E. Kelley Jr. da Remington Rand, para abordar desafios de planejamento e programação de projetos (Kelley Jr e Walker, 1961).

Ambas as técnicas, PERT e CPM, utilizam a representação gráfica de um projeto em forma de rede, onde os nós representam eventos e as ligações representam as atividades, como na Figura 10.4, que representa as atividades da Tabela 10.1.

Hoje em dia, porém, é comum representar de forma inversa, sendo o nó representando as atividades e as ligações as dependências entre elas (Kerzner, 2017), criando redes de precedência. Muitas vezes, nesse caso, são chamados apenas de **Diagramas de Rede**, como o mostrado na Figura 10.5, feita com o software MS Project. O projeto da Tabela 10.1 está apresentado na Figura 10.6, feita no software Project Libre, que é Open Source, mas menos configurável. Essas softwares normalmente também permitem a criação de diferentes tipos de dependência: a normal sendo fim para início, mas também

## 10. Gestão de Projetos de Software

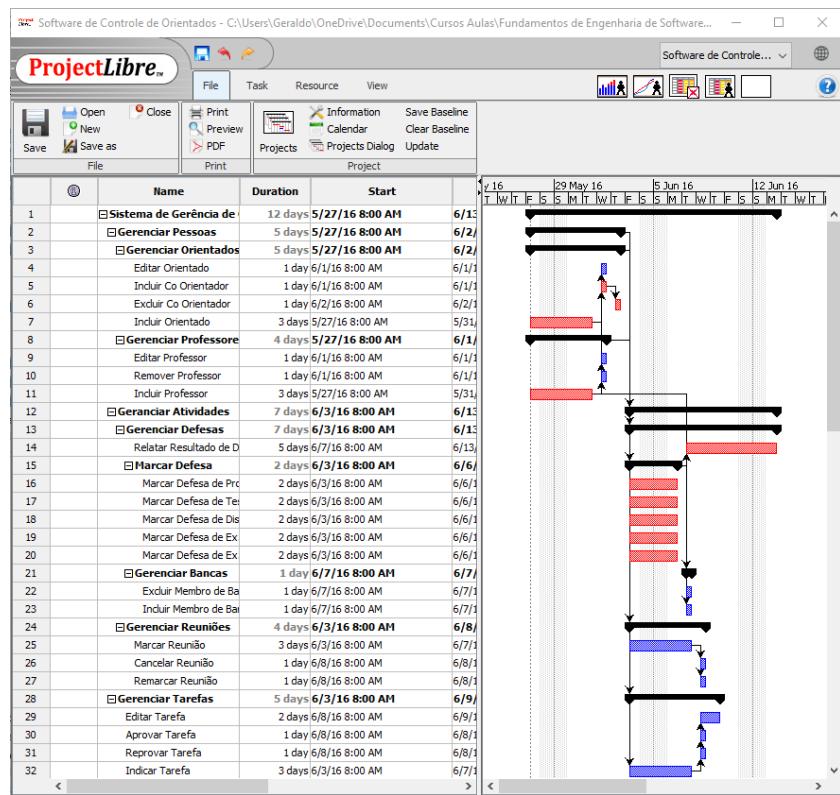


Figura 10.3.: Controlgrama de Gannt feito com o software Project Libre. Fonte: do autor

início com início, fim com fim, início com fim e até mesmo a partir de percentuais do trabalho completo Kerzner, 2017.

A distinção principal entre PERT e CPM reside no tratamento das estimativas de tempo. Enquanto o PERT usa três estimativas de tempo (otimista, mais provável e pessimista) para determinar uma estimativa esperada e um desvio padrão para as atividades, o CPM usa apenas uma estimativa determinística.

### Cálculo do Tempos a partir das Dependências

O principal uso dos diagramas de rede é calcular o tempo total do projeto, as folgas e o caminho crítico. O caminho crítico é o conjunto de atividades que se sofrerem atrasos levarão ao atraso do projeto.

A folga pode ser chamada de folga livre ou folga total. A folga livre é o tempo que uma atividade pode ser atrasada sem afetar o início da atividade imediatamente seguinte na sequência. A folga total é o tempo total que uma atividade pode ser atrasada sem atrasar a conclusão do projeto como um todo. Isso é, uma tarefa pode ultrapassar sua folga livre, mas mesmo assim não atrasar o projeto como todo. A explicação comum é que seu atraso vai ser compensando pela folga livre de uma outra atividade subsequente.

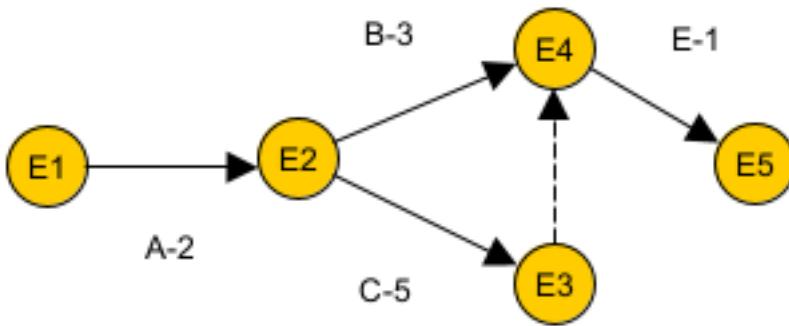


Figura 10.4.: Modelo PERT original para a Tabela 10.1

Tabela 10.1.: Tabela de tarefas de um projeto

Tarefa	Duração	Antecedentes
A	2	
B	3	A
C	5	A
D	1	B,C

Considerando um projeto com o tempo orçado em dias, dado um dia de início mais cedo de um projeto, é possível calcular a data de término mais cedo de um projeto simplesmente navegando pela rede e iniciando cada tarefa no dia seguinte ao que todas as tarefas de que dependem terminam. Por exemplo, considere as quatro tarefas e suas durações e dependências descritas na Tabela 10.1. Se a tarefa A começar no dia 1 de janeiro, então ela se encerrará no dia 2 de janeiro, perfazendo dois dias de trabalho. As tarefas B e C, que dependem de A acabar para começar, começarão no dia 3 de janeiro, e a tarefa B irá até o dia 5. A tarefa C, porém, é mais longa e irá até o dia 7. Assim, a tarefa D, que depende de B e C, só poderá começar dia 8. A tarefa B tem uma folga livre e total de dois dias. Se o projeto tiver que acabar no dia 8, as outras tarefas tem folga livre e total igual a zero. As tarefas A,B e D perfazem o caminho crítico.

Após o preenchimento das datas mais cedo de início e término das tarefas, é possível fazer a conta de volta, calculando as datas mais tarde e as folgas.

A partir da Tabela 10.2 podemos calcular, de trás para frente, as datas de término e início mais tarde das tarefas. Como decidimos que o projeto deve acabar na data de término mais cedo, a tarefa D fica com as datas do tipo mais tarde iguais as do tipo mais cedo. A diferença acontece na tarefa . A data de início mais tarde dela é o dia 7, já que a tarefa D, que depende dela, tem como data de início mais tarde o dia 8. Assim existe uma diferença entre a data de término mais tarde e a data de término mais cedo, logo ela tem uma folga livre de 2 dias. Já a tarefa B não tem folga, e também a tarefa A. O resultado pode ser visto na Tabela 10.3

## 10. Gestão de Projetos de Software

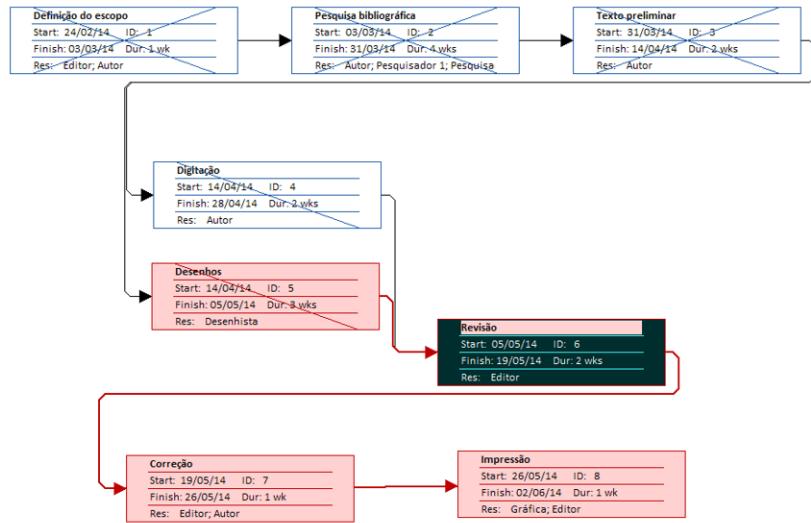


Figura 10.5.: Um diagrama de rede construído com o software MSProject, software altamente configurável. Vemos atividades completas, em andamento. Também é possível notar as atividades no caminho crítico. Fonte: do autor

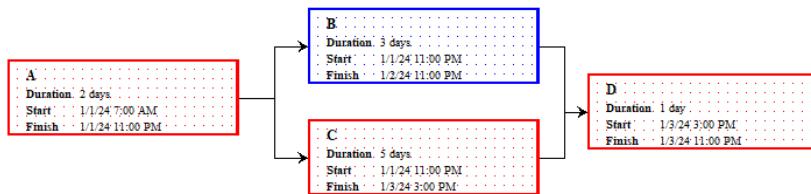


Figura 10.6.: Diagrama de rede para a Tabela 10.1, feito no software Project Libre. Fonte: o autor

O desenho de uma rede PERT/CPM permite fazer essas contas rapidamente. Para projetos com muitas atividades é recomendado usar um software dedicado.

### 10.1.4. Algoritmo para cálculo das datas em uma rede PERT/CPM

Para calcular as datas de início e término de atividades em uma rede de atividades, podemos seguir os seguintes passos:

#### Datas de Início Mais Cedo (IMC) e Término Mais Cedo (TMC)

1. Considere a primeira atividade com IMC = 1.
2. Para todas as atividades com o IMC , atualize o TMC da atividade como:

$$TMC = IMC + \text{duração da atividade} - 1$$

Tabela 10.2.: Tabela de tarefas de um projeto, com as datas de início e término mais cedo

Tarefa	Duração	Antecedentes	Início mais cedo	Término mais cedo
A	2		1	2
B	3	A	3	5
C	5	A	3	7
D	1	B,C	8	8

Tabela 10.3.: Tabela de tarefas de um projeto, com as datas de início e término mais cedo e mais tarde.

Tarefa	Duração	Antecedentes	Início mais cedo	Término mais cedo	Início mais tarde	Término mais tarde
A	2		1	2	1	2
B	3	A	3	5	5	7
C	5	A	3	7	3	7
D	1	B,C	8	8	8	8

3. Para todas as atividades subsequentes diretamente dependentes desta atividade que tiverem todas suas atividades dependentes com o TMC calculado, atualize o IMC delas como:

$$\text{IMC (atividade subsequente)} = \max(\text{TMC (atividades anteriores)}) + 1$$

onde o máximo é tomado sobre todas as atividades das quais a atividade subsequente depende.

4. Repita os passos 2 e 3 para todas as atividades até o fim da rede.

### Datas de Término Mais Tarde (TMT) e Início Mais Tarde (IMT)

1. Inicialize a atividade final com TMT igual ao seu TMC (determinado no passo anterior). Ou seja, o projeto terminará no TMC da última atividade.
2. Calcule o IMT da atividade final como:

$$\text{IMT} = \text{TMT} - \text{duração da atividade} + 1$$

3. Para todas as atividades anteriores diretamente relacionadas à essa atividade, atualize o TMT delas como:

$$\text{TMT (atividade anterior)} = \min(\text{IMT (atividades subsequentes)}) - 1$$

onde o mínimo é tomado sobre todas as atividades que dependem diretamente da atividade anterior.

## *10. Gestão de Projetos de Software*

4. Atualize o IMT da atividade anterior como:

$$\text{IMT} = \text{TMT} - \text{duração da atividade} + 1$$

5. Repita os passos 3 e 4 retrocedendo para todas as atividades até o início da rede.

As folgas de cada atividade podem ser calculadas como:

$$\text{Folga} = \text{IMT} - \text{IMC}$$

Uma folga de zero indica que a atividade está no caminho crítico.

## **10.2. Uso das Técnicas Tradicionais**

Hoje em dia, devido aos softwares disponíveis, como MSProject ou Project Libre, essas técnicas são todas usadas de forma conjunta, sendo possível passar de uma visão para outra com facilidade.

## **10.3. Técnicas Ágeis de Gestão de Projeto**

### **10.3.1. Kanban**

# **Parte IV.**

## **Especificação**



# 11

## Casos de Uso

We thus do not model reality as it is, as object orientation is often said to do, but we model the reality as we want to see it and to highlight what is important in our application

---

(Ivar Jacobson)

## Conteúdo

---

11.1.	Conceituação de Caso de Uso . . . . .	172
11.2.	Conceitos Importantes nos Casos de Uso . . . . .	173
11.3.	A Narrativa do Caso de Uso . . . . .	180
11.4.	O Nível de Abstração . . . . .	181
11.5.	O Escopo do Caso de Uso . . . . .	182
11.6.	Casos de Uso Especiais . . . . .	183
11.7.	Partes do Caso de Uso . . . . .	183
11.8.	Exemplo de Um Caso de Uso . . . . .	184
11.9.	Levantando Casos de Usos . . . . .	185
11.10.	Diagramas de Caso de Uso . . . . .	188
11.11.	O Que o Diagrama de Caso de Uso Não Informa . . . . .	195

## 11. Casos de Uso

### 11.12. Exercícios . . . . . 195

#### Por que casos de uso?

Junto com histórias do usuário, casos de uso são uma das principais formas de especificar requisitos, sendo ainda mais detalhadas que as primeiras. São usados em projetos ágeis ou prescritivos e um documento muito comum de se achar em fábricas de software.

Este capítulo apresenta Casos de Uso e Diagramas de Caso de Uso. É importante não confundir os dois conceitos. Um caso de uso é uma narrativa textual detalhada, acompanhada de informações que ajudam a entendê-la, normalmente apresentado na forma de um documento. Um diagrama de caso de uso é um gráfico que resume todos os casos de uso do sistema, não fornecendo nenhum detalhe, e que informa apenas uma visão global do escopo do sistema.

Casos de uso foram criados por Ivar Jacobson em 1986, aparecendo em um artigo da OOPSLA 87(Jacobson, 1987), mas só foram publicados em livro em 1992 (Jacobson, Christerson et al., 1992). Uma das melhores descrições de seu uso foi feita por Cockburn (2000), no livro *Writing Effective Use Cases*.

A parte principal de um caso de uso é uma sequência numerada de atividades que representam as ações que acontecem durante a interação de usuários com um sistema, como na descrição do cenário principal do caso de uso fictício “Avaliar Alunos” da Figura 11.6, que é um objetivo do ator professor e manda informações para o ator aluno.

1. Sistema Acadêmico apresenta tela de opções
2. Professor escolhe Entregar Notas
3. Sistema Acadêmico apresenta lista de cadeiras disponíveis
4. Professor escolhe cadeira
5. Sistema acadêmico apresenta lista de alunos com espaço para notas
6. Professor inclui notas para todos os alunos
7. Sistema Acadêmico registra notas
8. Sistema Acadêmico envia alteração de notas para todos Alunos por email
9. Sistema Acadêmico informa ao professor que as notas foram alteradas
10. Professor sai do Sistema Acadêmico

Figura 11.1.: Exemplo de cenário principal de caso de uso.

Casos de uso são uma das técnicas de maior sucesso na análise de sistemas, e apesar de serem identificados com a Análise Orientada a Objeto, não há nenhuma obrigação de serem usados apenas nesse contexto. A principal vantagem dos casos de uso sobre técnicas

anteriores é serem descrições do comportamento concreto do software a ser desenvolvido, quando no passado se dava muita importância a descrições abstratas que eram confusas para o usuário final e deixam grande margem de interpretação, e consequentemente eram ambíguas para o desenvolvedor (Jacobson, 2004).

Casos de uso contam como o sistema funciona, ou funcionará, na visão de seus usuários, o que é sempre uma visão externa ao sistema. Eles devem ser facilmente lidos tanto pelos clientes, e outras partes interessadas, quanto pelos desenvolvedores. São, na prática, histórias contada na forma de um passo a passo, de como o usuário usa o sistema e dos efeitos dessa utilização no estado do sistema.

Já um diagrama de caso de uso tem a aparência da Figura 11.2.

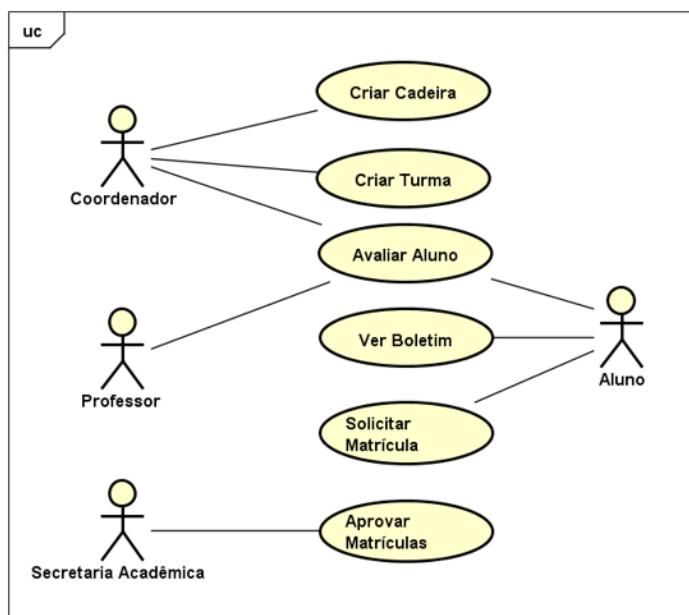


Figura 11.2.: Exemplo de um caso de uso.

Diagramas de caso de uso permitem ao seu leitor ter uma ideia do escopo do sistema. Eles apresentam os atores envolvidos e as funcionalidades esperadas do sistema de forma simples. Desde a origem da análise de sistemas é comum existir um diagrama cujo objetivo é mostrar o escopo do sistema, e o diagrama de caso de uso cumpre essa função na Análise Orientada a Objeto, sendo parte da especificação UMLOMG (2017). Basicamente eles substituíram os Diagramas de Contexto Gane e Sarson (1979) e Ruble (1997) nessa finalidade.

## 11.1. Conceituação de Caso de Uso

Um **caso de uso** é uma especificação, em forma de narrativa, de uma sequência de interações entre um sistema e os atores (agentes externos) que o usam de forma a alcançar um objetivo. Esse sistema pode ser um software ou uma organização complexa.

Cada caso de uso responde a uma solicitação de uma parte interessada, o **ator principal**, e descreve como o sistema vai se comportar enquanto interage com esse ator, de acordo com várias condições que podem ocorrer durante a interação(Cockburn, 2000).

Casos de uso podem ser simples ou complexos, devendo descrever, em um nível de detalhe desejado, algo que um usuário ou cliente quer que o sistema faça. Cada caso de uso descreve e define parte da funcionalidade de um sistema.

As definições de caso de uso, pelos seus criadores, evoluíram com o tempo. Jacobson, Spence e Bittner (2011) citam duas definições, que são apresentadas a seguir.

- Um **caso de uso** mostra todas as formas de usar um sistema para alcançar um objetivo específico para um usuário específico. O conjunto de casos de uso fornece todas as formas úteis de usar o sistema e obter o valor que ele oferece.
- Um **caso de uso 2.0** é uma prática ágil e escalável que captura um conjunto de requisitos e guia o desenvolvimento incremental de um sistema para atingi-los.

Um caso de uso bem escrito deve descrever, de forma completa, um processo executado pelo sistema, contando uma história de como o ator principal tenta alcançar um objetivo específico ao usar o sistema. Na sua forma mais completa, um caso de uso apresenta vários cenários possíveis de sucesso ou falha na busca por esse objetivo(Cockburn, 2000) e ainda informações adicionais para o time.

O conjunto completo de casos de uso forma a especificação funcional de um sistema. Essa especificação é facilmente legível por usuários ou desenvolvedores, permitindo também sua validação e verificação. É importante notar que os diagramas de caso de uso têm um valor muito pequeno frente ao documento textual que é a descrição do caso de uso.

Um caso de uso deve:

- descrever um tarefa de negócio que serve a um único objetivo de negócio;
- não ser orientado a uma linguagem de programação;
- ter o nível de detalhe apropriado;
- ser curto o suficiente para ser implementado por um desenvolvedor de software em um versão do produto;
- ser descrito do ponto de vista externo, e
- ser consistente, tanto no nível de abstração quanto na escolha entre mostrar o sistema como uma caixa branca ou uma caixa preta.

Casos de uso também podem ser vistos como “contratos entre as partes interessadas e o sistema sobre o seu comportamento” Cockburn (2000).

Para isso, casos de uso devem descrever três tipos de ações(Cockburn, 2000):

- **interações entre dois atores**, sendo um deles normalmente o sistema;
- **validações**, que protegem uma ou mais partes interessadas, e
- **mudanças do estado interno do sistema**, que atendem uma ou mais partes interessadas.

## 11.2. Conceitos Importantes nos Casos de Uso

Os principais conceitos envolvidos em um caso de uso são:

- **atores**, e entre eles o **ator principal**, que participam do caso de uso;
- um **objetivo** do ator principal, que nomeia o caso de uso;
- os **cenários** que permitem atingir o objetivo ou levam a falhas, entre eles o **cenário principal** e vários possíveis **cenários alternativos**;
- as **condições** que geram os cenários alternativos;
- cada cenário é um **sequência de ações** que acontece durante a interação entre atores e o sistema, e
- cada ação é:
  - uma troca simples e unidirecional de informações entre dois atores;
  - uma validação, ou
  - uma mudança do estado interno do sistema.

### 11.2.1. Ator

Um ator é uma entidade externa ao sistema com comportamento próprio. Com isso queremos dizer que o sistema não pode controlar o comportamento do ator.

Os atores interagem com o sistema para alcançar seus objetivos. Em cada Caso de Uso, um ator é o ator principal, que visa alcançar com sucesso o seu objetivo principal por meio daquele caso de uso. Muitas vezes, o sistema invocará outros atores, que deverão cumprir suas responsabilidades para o ator principal alcançar seu objetivo, ou receberão informações.

Um ator não é a mesma coisa que um usuário. Um ator representa um papel dos usuários de um sistema. Tanto um ator pode representar um papel assumido por vários usuários, como o papel de correntista em um banco, para o qual existem milhares de usuários, quanto um usuário (pessoa real) pode ser representado por vários atores, como no caso de um funcionário do banco que é também correntista. Conceitualmente, em UML, usuários são instâncias de atores e cada ator define uma classe de usuários(OMG, 2017).

Os principais tipos de atores são:

- pessoas;

## 11. Casos de Uso

- organizações;
- equipamentos, e
- sistemas.

Um ator, nos diagramas de caso de uso, é representado por um “boneco de pauzinhos” sobre o nome que identifica o ator, como na Figuras 11.3 e Figura 11.4.



Figura 11.3.: Representação de um ator em um diagrama de caso de uso.



Figura 11.4.: Representação de dois atores identificados.

**O sistema sendo descrito também é um ator**, que conversa com o ator principal e com os outros atores.

É importante notar que existem partes interessadas que não são atores, pois nunca interagem com o sistema. Por exemplo, o patrocinador ou os próprios desenvolvedores, mas ainda assim podem ser afetadas pelas ações do sistema. Um exemplo possível seria um sistema de segurança pública que guarda informação sobre criminosos, mas esses criminosos nunca tem contato com o sistema.

### 11.2.2. Objetivos

Todo ator possui um ou mais objetivos ao usar o sistema. Alguns objetivos são ativos, isso quer dizer que o usuário invoca o sistema, se tornando o ator principal de um caso de uso, e outros podem ser ditos passivos, ou seja, o sistema vai informar o ator de algo ou invocar o ator para alguma ação.

Cada objetivo define e dá nome a um caso de uso. Esse objetivo deve deixar bem claro o que o ator deseja obter como funcionalidade do sistema, definindo de forma implícita o comportamento que ele espera do sistema.

Exemplos de objetivos são:

## 11.2. Conceitos Importantes nos Casos de Uso

- em um sistema bancário, depositar dinheiro em conta corrente, sacar dinheiro de conta corrente, pagar boleto, transferir dinheiro para outra conta corrente, requisitar cheques;
- em um sistema de seleção de profissionais, enviar currículo, marcar entrevista, avaliar currículo, solicitar classificação dos candidatos, e
- em um sistema acadêmico, solicitar matrícula, avaliar alunos, informar presenças, emitir boletim.

### Nomeando Casos de Uso

O objetivo é usado para nomear o caso de uso. Isso é feito no formato

*<verbo no infinitivo><objeto>*

como em “emitir boletim”. Para isso devem ser usados verbos que indiquem uma ação do ator, pois ele entra no sistema com a intenção de fazer algo. A Tabela 11.1 apresenta alguns verbos comumente usados para nomear casos de uso.

Tabela 11.1.: Verbos comumente usados para nomear casos de uso

Alterar	Descobrir	Permitir
Analisar	Encontrar	Preencher
Aprontar	Entregar	Preparar
Arranjar	Especificar	Projetar
Avaliar	Estabelecer	Providenciar
Buscar	Executar	Realizar
Classificar	Garantir	Recuperar
Completar	Identificar	Requisitar
Conseguir	Informar	Selecionar
Consultar	Monitorar	Solicitar
Definir	Mudar	Ver
	Notificar	

### 11.2.3. Cenários

Um caso de uso pode acontecer de acordo com vários cenários. Cada **cenário** descreve como uma instância específica do caso de uso pode acontecer, ou seja, as sequências específicas de ações que ocorrem na interação entre atores e o sistema.

Um desses cenários é o **cenário principal**, que narra como um ator alcança seu objetivo da forma mais fácil ou comum. O cenário principal é descrito de forma integral em todos os casos de uso e mostra como é esperado que a interação normal que atinge o objetivo aconteça. Por exemplo, para o caso de uso “sacar dinheiro”, solicitado pelo

## 11. Casos de Uso

ator “correntista” para o sistema “caixa automático”, o cenário principal poderia ser o da Figura 11.5:

1. correntista toca na tela
2. caixa automático solicita cartão
3. correntista passa o cartão
4. caixa automático valida cartão
5. caixa automático solicita senha
6. correntista informa senha
7. caixa automático valida senha
8. caixa automático apresenta menu
9. correntista escolhe sacar da conta corrente
10. caixa automático solicita valor
11. correntista digita valor
12. caixa automático valida existência de saldo
13. caixa automático solicita cartão
14. correntista passa o cartão
15. caixa automática valida cartão
16. caixa automático fornece o dinheiro
17. caixa automático informa banco para debitar valor do saldo bancário do correntista
18. correntista retira o dinheiro
19. caixa automático fecha a sessão

Figura 11.5.: Exemplo de cenário principal do caso de uso **sacar dinheiro**.

O cenário acima é descrito na forma de uma **narrativa numerada**. Cada passo dessa narrativa é uma troca simples, e unidirecional, de informações, ou de uma mensagem, uma validação ou uma ação. Essa narrativa é o que deve acontecer se tudo der certo, ela não se preocupa, por exemplo, com o que fazer se não há saldo para retirar a quantia ou se a senha está errada. Também não se preocupa se o usuário quer fazer outra ação, como imprimir cheques.

Veja que no passo 17 o caso de uso envolve um outro ator, o banco, para o qual é enviada uma mensagem de alteração de saldo. Uma alternativa a esse passo seria “caixa automático debita valor do saque do saldo do correntista”. Esse é um tipo de passo que indica uma mudança no estado interno do sistema e também devemos considerar obrigatório, pois indica o contrato que o caso de uso cumpre.

Casos de uso também apresentam passos de validação, que tem como função proteger os interesses das partes interessadas. Os passos 4, 7, 12 e 15 fazem a validação e serão usados como locais onde condições definem alternativas.

Cenários, além do cenário principal, podem representar variantes normais do fluxo principal, casos raros, exceções e erros. Dessa maneira, podemos compreender um caso de uso como a descrição de uma coleção de cenários de sucesso ou falha que descrevem um determinado processo do sistema com a finalidade de atender um objetivo do usuário.

## 11.2. Conceitos Importantes nos Casos de Uso

Na maior parte das vezes, os casos de uso também possuem vários **cenários alternativos**, alguns que também levam ao objetivo desejado, outros que levam a versões parciais desse objetivo e ainda cenários de falhas, onde o objetivo não é alcançado. Todos esses cenários também são descritos nos casos de uso, porém normalmente de forma compactada, sendo mostrado apenas como eles diferem do caso de uso principal.

Os cenários diferem em função de condições específicas que podem acontecer na execução de uma instância do caso de uso. Por exemplo, se o caso de uso descreve uma retirada de dinheiro de uma conta bancária, o cenário principal considera que existe saldo suficiente na conta, enquanto cenários alternativos podem considerar que não existe saldo suficiente, que será necessário usar o cheque especial, ou que é tarde demais para retirar a quantia pedida.

No nosso exemplo, uma condição que leva a um cenário alternativo poderia ser poderia ser “senha não confere” e outra poderia ser “não há saldo suficiente”. Nesse caso, cada condição levaria a um cenário alternativo.

Uma forma possível de descrever o cenário alternativo “senha não confere” seria:

1. caixa automático informa que senha não confere
2. voltar ao passo 5

Nesse caso ainda é possível terminar o caso de uso principal, o sistema volta para um passo anterior ao que aconteceu a alternativa e o ator tem mais uma chance.

Uma condição adicional seria “senha não confere pela terceira vez”, e o cenário relativo seria:

1. caixa automático informa que senha não confere
2. caixa automático informa ao banco para bloquear o cartão
3. caixa automático termina atendimento

Nesse caso não foi possível voltar ao cenário principal, o ator não conseguiu seu intento e o caso de uso terminou em falha.

Novamente, nesse último cenário, teríamos a opção de que o passo 2 fosse “caixa automático bloqueia cartão”, que indicaria uma mudança no estado do sistema.

As Figuras 11.6 e 11.7 ilustram o conceito de um caso de uso contendo um caminho principal e quatro condições que podem alterar a execução de uma instância do caso de uso. As figuras mostram, de forma abstrata, possíveis caminhos que formam cenários de execução desse caso de uso. Podemos ver, na representação da figura, que alguns caminhos alternativos podem permitir ainda alternativas adicionais (o caminho 2 é uma alternativa do caminho 1), e que em uma mesma execução de um caso de uso vários caminhos podem ser seguidos (os caminhos 1 e 3, por exemplo).

Com essas figuras deve ficar claro um conceito: os cenários alternativos podem ser compostos de várias formas, mas com uma descrição compartimentada, não precisamos usar todas as instâncias possíveis dos cenários, por exemplo listar todos os cenários possíveis onde a alternativa 3 é chamada 1, 2, 3 ou infinitas vezes.

## 11. Casos de Uso

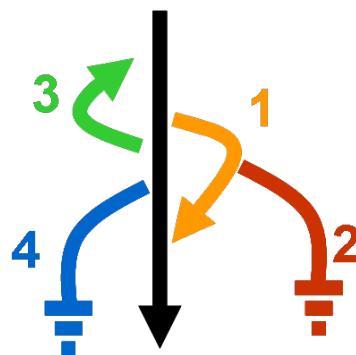


Figura 11.6.: Representação abstrata de um cenário principal e dos fluxos alternativos

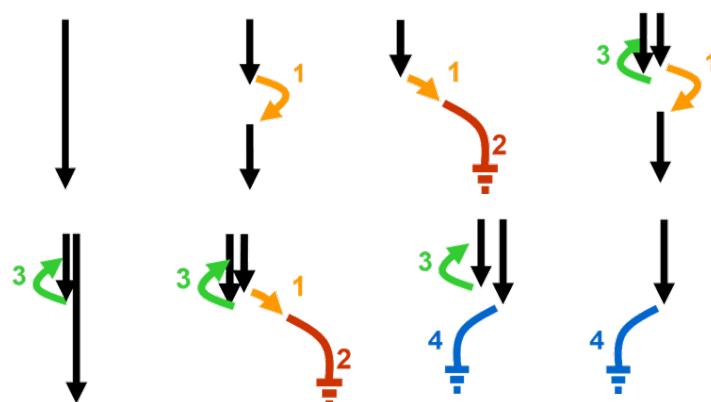


Figura 11.7.: Representação abstrata de cada alternativa investigada nesse caso.

Também é importante notar que **não são usadas estruturas se-então-senão** ao escrever casos de uso. O que se usa são condições que podem acontecer em passos de um cenário e ordens como “vá para”. Cada cenário considera que todos seus passos de validação resultam em verdade.

Já o uso de laços, isto é, estruturas de repetição, pode ser feito de algumas formas, mas nunca na forma de um *for* ou *while* de uma linguagem de programação, mas sim como uma linguagem mais coloquial. (Cockburn, 2000) sugere:

- usar um passo onde algo pode ser feito várias vezes, como em “o comprador seleciona um ou mais produtos”, ou
- um passo não numerado que indica que alguns passos podem ser repetidos, como em “o comprador repete os passos 4-6 até ficar indicar que terminou”, que pode aparecer antes ou depois dos passos;

Por exemplo, um caso de uso onde o ator “professor” dá a nota para vários alunos poderia ser do da Figura 11.8.

A Tabela 11.2 mostra uma notação alternativa para explicar os cenários possíveis a partir de alternativas.

## 11.2. Conceitos Importantes nos Casos de Uso

1. professor seleciona DAR NOTAS  
os passos 2-7 acontecem até o professor indicar que terminou
2. sistema mostra lista de alunos
3. professor escolhe aluno
4. sistema mostra registro do aluno
5. professor dá nota ao aluno
6. sistema altera a nota do aluno
7. sistema pergunta se o professor terminou
8. professor sai do sistema

Figura 11.8.: Exemplo de cenário principal do caso de uso **atribuir notas**.

Tabela 11.2.: Uma forma alternativa de representar os cenários possíveis a partir de um cenário principal e as alternativas associadas

<b>Cenário 1</b>	Cenário Principal			
<b>Cenário 2</b>	Cenário Principal	Alternativa 1		
<b>Cenário 3</b>	Cenário Principal	Alternativa 1	Alternativa 2	
<b>Cenário 4</b>	Cenário Principal	Alternativa 3		
<b>Cenário 5</b>	Cenário Principal	Alternativa 3	Alternativa 1	
<b>Cenário 6</b>	Cenário Principal	Alternativa 3	Alternativa 1	Alternativa 2
<b>Cenário 7</b>	Cenário Principal	Alternativa 4		
<b>Cenário 8</b>	Cenário Principal	Alternativa 3	Alternativa 4	

### 11.2.4. As Alternativas

As alternativas estão sempre associadas a uma condição ocorrendo no cenário principal, possivelmente em um momento de validação, ou, mais raramente, a um outro cenário importante.

Um cenário alternativo tem pelo menos quatro partes:

- a localização, mostrando o cenário, normalmente o principal, e o passo onde ocorre a condição;
- a condição, especificada na forma de uma expressão que pode ser avaliada como verdadeira ou falsa (lógica);
- a sequência de ações que corresponde a essa condição, e
- uma finalização que deixa claro o retorno ao fluxo principal ou a finalização de outra forma.

As finalizações possíveis para uma alternativa são:

- voltar ao início do caso de uso, comum em casos de falha onde é possível tentar outra vez;

## 11. Casos de Uso

- voltar a um passo já executado do caso e uso, trazendo o estado do sistema naquele passo;
- ir para um passo posterior ao passo onde ocorreu a condição, possivelmente o final;
- encerrar o caso de uso no cenário alternativo, com sucesso ou sucesso parcial;
- encerrar o caso de uso no cenário alternativo com falha, e
- abortar o caso de uso, em caso de falha grave.

### 11.2.5. Casos de Uso em UML

Em UML casos de uso descrevem o comportamento de um sistema, ou parte de sistema, como um componente ou classe, em consideração, denominado *subject*. Um caso de uso pode se aplicar a vários *subjects*, especificando um conjunto de comportamentos para os *subjects* a que é aplicado. Os atores são pessoas e outros sistemas que interagem com o *subject*. Um caso de uso gera um resultado que tem valor para os atores e outras partes interessadas no *subject*(OMG, 2017).

## 11.3. A Narrativa do Caso de Uso

A narrativa numerada é a forma de básica de descrição dos cenários do caso de uso. Essa forma de narrativa apresenta várias vantagens sobre outras formas já utilizadas de modelagem de processos, como manter o contexto visível, eliminar dificuldades de compreensão para o usuário (causadas por linguagens técnicas com forte grau de abstração) e deixar claro qual o valor de cada função para os usuários.

Cada caso de uso possui uma narrativa principal, que mostra o que acontece no caso normal, e narrativas alternativas que tratam de atender a condições especiais que acontecem durante uma possível execução do caso de uso e fazem ele se afastar da narrativa principal.

Essa narrativa é uma especificação de uma sequência de ações que acontece interações entre o sistema e os agentes externos que usam esse sistema.

Segundo Cockburn (2000), cada passo da narrativa deve:

- usar uma gramática simples;
- mostrar quem está agindo;
- mostrar o processo andando para frente;
- mostrar a intenção do ator, não os movimentos, e
- conter um conjunto de ações razoável;

Outras formas de escrever o caso de uso são usadas no mercado, sendo que é comum que no início da análise seja usado um texto informal na forma de um parágrafo, muitas vezes tirado diretamente de uma entrevista.

## 11.4. O Nível de Abstração

Uma das formas mais interessantes de entender como desenvolver casos de uso é entender que eles podem ser descritos em diferentes níveis de abstração, desde um nível bastante abstrato até um nível detalhado em seus mínimos detalhes.

A idéia básica em torno desse conceito, proposto por Cockburn (2000), é que casos de uso de um nível mais abstrato explicam o porquê de um caso de uso de um nível mais baixo, enquanto o caso de uso de um nível mais baixo explica como o caso de uso do nível mais abstrato é realizado.

Podemos identificar cinco níveis de abstração para casos de uso (Cockburn, 2000):

- sumário de alto nível;
- sumário;
- objetivo do usuário;
- sub-função, e
- muito detalhado.

Tabela 11.3.: Ícones que representam o nível de abstração dos casos de uso.

Fonte:(Cockburn, 2000)

Nível	Ícone
Sumário de Alto Nível	
Sumário	
Objetivo do Usuário	
Sub-Função	
Nível Muito Detalhado	

A abordagem adotada neste texto é trabalhar no nível de objetivo do usuário. Se necessário, para projetos grandes, é possível criar resumos no nível de sumário. Por exemplo, os casos de uso “sacar dinheiro”, “depositar dinheiro” e “transferir dinheiro” poderiam ser resumidos em um caso de uso de nível superior chamado “movimentar conta”.

Raramente são usados casos de uso mais detalhados do que no nível do objetivo do usuário, a não ser os casos de uso incluídos, que são normalmente do tipo sub-função.

## 11.5. O Escopo do Caso de Uso

Da mesma forma que definimos o nível de abstração de um caso de uso, podemos também definir o escopo do caso de uso em relação a organização ou sistema que ele descreve em níveis (Cockburn, 2000)

- organização, caixa preta;
- organização, caixa branca;
- sistema, caixa preta;
- sistema, caixa branca, e
- componente.

Tabela 11.4.: Ícones que representam o nível de escopo dos casos de uso. Fonte:(Cockburn, 2000)

Nível	Ícone
Organização, caixa preta	
Organização, caixa branca	
Sistema, caixa preta	
Organização, caixa branca	
Componente	

O nível que trabalhamos nesse texto é normalmente o nível de sistema de caixa preta. Nesse nível, tratamos de atores nomeados usando um sistema que é sempre chamado por seu nome.

No nível de sistema com caixa branca, passamos a nomear a parte do sistema que está sendo usada.

No nível organização e caixa preta, em vez de tratar do uso do sistema, se trata da interação do usuário com a organização. Assim, um correntista passa a conversar com o banco, um paciente com o hospital. No caso da caixa branca, o correntista passa a falar com o caixa ou com o gerente, o paciente com o médico ou a enfermeira. Também é possível usar a caixa branca em relação a setores da organização, como tesouraria e atendimento.

## 11.6. Casos de Uso Especiais

Alguns casos de uso não tem origem nos usuários ou cliente, mas em outras partes interessadas que precisam implantar e operar o sistema. Esses casos de uso são especiais e podem ser usados poucas vezes, mas podem ser bastante complexos. Alguns casos possíveis são:

- início do sistema;
- parada do sistema;
- manutenção da informação, inclusive com operações do tipo CRUD adicionais;
- adicionar nova funcionalidade com o sistema funcionando, em especial para sistemas que não devem parar;
- transferência do sistema para um novo ambiente, etc.

**Create, Read, Update e Delete,**  
operações que sempre podem ser realizadas sobre os dados mesmo sem ter uma semântica do negócio

## 11.7. Partes do Caso de Uso

Um caso de uso completo é um documento com várias partes. Entre as partes possíveis que podemos usar estão:

- nome;
- identificação(número, etc.);
- escopo e nível de abstração(Cockburn, 2000);
- ator principal;
- outros atores;
- outras partes interessadas;
- pré-condições;
- gatilhos;
- pós-condições, em caso de sucesso;
- garantias, em caso de falha;
- cenário principal;
- cenários alternativos;
- requisitos especiais;
- variações tecnológicas ou de dados;
- justificativa;
- fonte ou origem;
- casos de uso relacionados;
- questões em aberto.

Já conhecemos a importância dos atores. A seguir discutiremos as outras partes do documento.

As **partes interessadas** a serem tratadas em um caso de uso são aquelas que não fazem parte do caso, mas podem ter interesse em seu resultado. Por exemplo, em um laboratório de exames clínicos, se um ator “técnico em análises clínicas” usa o caso de

## 11. Casos de Uso

uso “registrar resultado de exame”, o ator “paciente” é uma parte interessada, mesmo que só receba o resultado após o ator “médico” liberar o resultado.

As **pré-condições** de um caso de uso definem o que é sempre verdadeiro quando um caso de uso acontece. São condições que **não precisam ser testadas**, ou seja, são assumidas como verdadeiras no caso de uso. Um exemplo no caso de uso “registrar resultado de exame” seria “exame já está cadastrado”. Só devem ser comunicadas nessa seção questões dignas de nota, que constituam informação útil para o desenvolvimento e funcionamento do sistema.

Os **gatilhos** são eventos, como ações ou condições, que causam o início do caso de uso. Origem ou fonte registra quem ou o que, como um documento, demandou ou indicou a necessidade do caso de uso.

As **pós-condições** estabelecem o que deve ser verdadeiro após o caso de uso, no caso de sucesso, por exemplo garantir que caso um caixa automático tenha dado o dinheiro ao correntista que esse dinheiro tenha sido debitado da conta do mesmo. As **garantias** são o estado final caso haja um erro, por exemplo não alterando o saldo do correntista caso ele não tenha recebido o dinheiro.

Pré-condições, pós-condições e garantias são uma especificação do estado anterior e posterior do sistema e servem como parte do contrato do caso de uso.

Justificativa indica a motivação, o interesse para aquele caso de uso.

Requisitos especiais estão relacionados a demandas adicionais que são feitas a esse caso de uso específico e que não são gerais do sistema. Por exemplo, tecnologias específicas que devem ser usadas, velocidade, etc.

### 11.8. Exemplo de Um Caso de Uso

Caso de Uso: 1	Solicitar Boletim Oficial
<i>Escopo:</i>	Sistemas, caixa-branca
<i>Nível de abstração:</i>	Objetivo do usuário
<i>Ator principal:</i>	Aluno
<i>Ator secundário:</i>	Secretaria Acadêmica
<i>Outras partes interessadas:</i>	<ul style="list-style-type: none"><li>• Coordenador: Tem que assinar o documento</li></ul>

*Pré-condições:*

- O aluno está logado no sistema
- O aluno tem notas em ao menos um período

---

*Pós-Condições:*

O pedido do aluno está incluído na lista de tarefas da Secretaria Acadêmica

---

*Gatilho:*

Aluno escolhe solicitar boletim oficial no menu

---

*Cenário Principal:*

1. Sistema pede para o aluno confirmar o pedido
  2. Aluno confirma o pedido
  3. Sistema verifica se o aluno não tem nenhum pedido na fila
  4. Sistema coloca o pedido do aluno na fila de tarefas da Secretaria Acadêmica de seu curso
  5. Sistema avisa a Secretaria Acadêmica que há uma nova tarefa na fila
  6. Sistema informa ao aluno que o pedido foi completado
  7. Aluno aceita a mensagem
- 

*Alternativas:*

3.a Aluno já tem um pedido na fila:

1. Sistema avisa Secretaria Acadêmica que um pedido foi refeito
  2. Sistema avisa aluno que seu pedido já estava na fila e que um aviso foi dado a Secretaria Acadêmica
  3. Aluno aceita a mensagem
- 

*Requisitos Especiais:* O sistema deve responder ao aluno em até 5s

---

*Variações de tecnologia:* Essa função deve funcionar na versão para telefone celular

---

*Assuntos em Aberto:* Será necessário comprovar o pagamento de uma taxa?

---

## 11.9. Levantando Casos de Usos

O processo de levantar casos de uso é um processo investigativo e iterativo. Ele pode ser resumido em 6 passos, que devem ser feito iterativamente até alcançar a especificação desejada:

1. identificar os atores;
2. identificar os objetivos de cada ator, nomeando seus casos de uso;
3. escrever o cenário principal para cada objetivo;
4. determinar as condições que geram as alternativas nos cenários existentes;

## 11. Casos de Uso

5. escrever o cenário alternativo para cada condição, e
6. detalhar as variações de dados.

Para identificar os atores é necessário investigar não só os atores que são pessoas físicas, mas também que sistemas, subsistemas, organizações, grupo de pessoas vão interagir com o software. Muitas vezes um caso de uso que começa com um ator precisa se comunicar de alguma forma com outro ator, por exemplo, enviando mensagens. O resultado do primeiro passo é uma lista de atores, que poderá ser estendida nos próximos passos.

Sempre que um ator novo for encontrado, durante a elicitação de um caso de uso, ele deverá ser investigado para que se possa avaliar se ele também vai demandar casos de uso, isto é, se terá objetivos para usar o sistema.

Atores devem ser nomeados com um nome específico do negócio ou da área de aplicação. Assim um sistema acadêmico possui aluno, professor, coordenador, etc. Um consultório ou ambulatório possui enfermeiro, auxiliar de enfermagem, médico, paciente, atendente, acompanhante, etc. Uma loja de roupas possui cliente, estoquista, gerente, vendedor, etc. Usar nomes genéricos, como usuário, demonstra má qualidade do modelo. O resultado do primeiro passo é uma lista de atores, que responde a pergunta “quem?”.

Para identificar o objetivo é importante entender o que cada ator precisa do sistema, quais suas necessidades e como elas devem ser demandadas ao sistema. O resultado do segundo passo é uma lista de casos de uso, que responde a pergunta “o que?” e um diagrama de casos de uso, que fornece uma visão geral do escopo do sistema. Com isso é possível ter uma lista delimitada e usável das funções do sistema.

Objetivos são escritos no formato verbo-objeto. Alguns exemplos são: solicitar boletim (para o ator aluno usando um sistema acadêmico) e lançar notas (para o ator professor no mesmo sistema).

No terceiro passo a função do analista de sistemas é determinar como o usuário alcançará seu objetivo usando o sistema. Esse passo só deve ter as mensagens trocadas entre os agentes (atores e sistema). Não **pode ser usada nenhuma estrutura de controle**, como laços e decisões. O cenário deve capturar a intenção e responsabilidade de cada ator até alcançar o objetivo. Cada passo do cenário deve deixar bem clara a informação que é passada entre os atores. O resultado é uma descrição legível das funções do sistema na forma que seriam executadas se tudo ocorresse normalmente, sem nenhuma condição especial ser ativada.

Um cenário principal para um Sistema Acadêmico, que descreve o caso de uso “emitir boletim”, para atender o ator “aluno”, poderia ser descrito, nessa fase, como na Figura 11.9:

O quarto passo usa como referência o cenário principal, e outros cenários alternativos do caso de uso em uma interação mais avançada, determinando as alternativas a partir dos passos já descritos. Cada passo pode ter alternativas que levem ao fracasso, ao sucesso parcial ou ao sucesso total. Cada condição deve ser anotada, gerando uma lista

1. aluno informa DRE
2. sistema valida DRE
3. sistema acadêmico solicita senha
4. aluno informa senha
5. sistema acadêmico valida senha
6. sistema acadêmico apresenta menu
7. aluno escolhe emitir boletim
8. sistema acadêmico emite boletim
9. aluno sai do sistema

Figura 11.9.: Exemplo de cenário principal do caso de uso **emitir boletim**.

de cenários alternativos que indica o cenário, o passo e a condição que pode ocorrer nesse passo.

No exemplo anterior, a lista de condições poderia ser:

- DRE não é validado
- senha não é validada
- se passam 2 minutos sem o aluno fazer uma ação

No quinto passo são resolvidas todas as alternativas. Isso é feito escrevendo o cenário, seguindo sempre as mesmas regras do cenário principal, e também indicando, ao final do cenário, se foi sucesso, parcial ou total, ou falha, e se é necessário voltar ao fluxo original.

A técnica do caso de uso descreve as alternativas apenas no que elas são diferentes do cenário principal, isto é, elas são normalmente menos passos do que o cenário principal e se referem a ele. O resultado dessa fase é uma versão completa dos casos de uso do sistema.

Finalmente, no último passo, são tratadas as questões que são de baixo nível para a análise, mas precisam ser resolvidas para a implementação. Por exemplo, como será emitido o boletim, que dados serão usados, etc.

Um bom caso de uso corresponde a um processo elementar da organização. Ele descreve a interação de um ator principal durante uma sessão única, onde um objetivo é atingido. Essa sessão **não** leva dias ou se divide em várias utilizações do sistema. Ele deve ser uma conversação, e não um passo único como “apagar um item”. Deve ser uma tarefa, concluída em uma sessão e que produz um resultado mensurável e partindo de uma situação onde as informações estão em um estado consistente e terminando em uma situação onde as informações também estão em um estado consistente, possivelmente diferente do estado inicial.

Casos de uso devem ser pensados como os motivos atômicos, isto é, mais simples, que o usuário tem para usar o sistema. Assim, em um sistema acadêmico, um ator “aluno” pode ter como objetivo de entrar no sistema “emitir o boletim” ou “fazer matrícula em cadeira”. É claro que um aluno específico pode entrar no sistema para fazer essas duas coisas, porém no processo de análise de sistemas elas são tratadas em separado, porque

## 11. Casos de Uso

é possível entrar no sistema para fazer essas atividades, mas dentro delas não há nada menor para ser feito, logo elas são atômicas.

O mesmo exemplo pode ser dado com um correntista de banco. Ele pode usar o caixa automática para sacar dinheiro, ou para emitir cheques, logo esses são casos de uso. Ele também pode fazer as duas coisas em uma sessão, mas como essa atividade seria uma composição de outras duas que podem ser feitas isoladamente, isto é, podem ser um objetivo único de um correntista, então as duas atividades isoladas são casos de uso, mas a atividade composta não. Além disso, não faz sentido um usuário entrar no sistema só para fazer um passo menor do que isso, como “entrar senha”.

A lista a seguir mostra algumas perguntas que podem ajudar a eliciar os casos de uso.

- Quais são as tarefas de um ator no negócio?
- O ator precisa ser informado de que ocorrências que ocorrem no sistema?
- O ator precisa informar o sistema de mudanças no ambiente externo?

Cockburn (2000) propõe uma receita mais detalhada, com 12 passos:

1. encontre as fronteiras do sistema, usando um diagrama de contexto ou uma lista de entradas e saídas;
2. faça um *brainstorm* e list os atores primários, gerando a **lista de atores**;
3. faça um *brainstorm* e gere os objetivos principais dos atores perante o sistema, gerando uma lista de atores e objetivos;
4. escreva os casos de uso no nível de sumário de todos os casos encontrados;
5. reconsidere e revise os casos de uso estratégicos, adicione, retire, junte e separe objetivos;
6. escolha um caso de uso para expandir ou escreva narrativa para ficar acostumado com o material;
7. preencha as partes interessadas, interesses, pré-condições e garantias, e verifique esse trabalho;
8. escreve o cenário principal de sucesso, verifique frente os interesses e garantias;
9. faça um *brainstorm* e liste as principais condições de sucesso e falha alternativas;
10. escreva como os atores e os sistemas devem se comportar em cada alternativa;
11. extraia qualquer sub caso de uso que se mostre necessário, e
12. inicie do início e reajuste os casos de uso, adicione, retira, junte ou separe casos de uso, verifique para completude, legibilidade e condições de falha.

### 11.10. Diagramas de Caso de Uso

Um diagrama de caso de uso representa de forma muito abstrata todos os casos de uso de um sistema. Seus principais componentes são atores e casos de uso.

Um diagrama de caso de uso pode ser construído com apenas 3 figuras básicas: atores, casos de uso e associações, apresentadas na Figura 11.10,

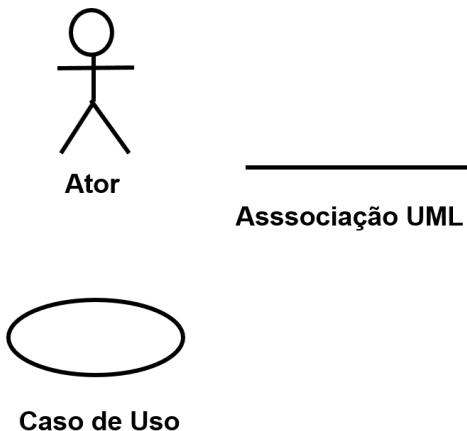


Figura 11.10.: Os elementos básicos de um diagrama de caso de uso são: atores, casos de uso e associações. Fonte: do autor

Outros símbolos, característicos dos casos de uso, também existem, representando relações entre atores e casos de uso. Esses símbolos são: herança, inclusão e extensão.

Finalmente, símbolos genéricos de UML como sistema/partição, comentário, etc., também podem ser usados.

Atores e casos de uso podem se relacionar de várias maneiras, como é mostrado na Tabela 11.6.

Tabela 11.6.: Associações que podem ocorrer entre atores e casos de uso.

	<b>ator</b>	<b>caso de uso</b>
<b>ator</b>	generalização	associação
<b>caso de uso</b>	associação	«include» «extend» generalização

### 11.10.1. Especificando o *Subject*

De acordo com o padrão UML(OMG, 2017), é possível indicar a que sistema caso de uso se refere, o que o padrão chama de *subject*, por meio de uma caixa, como feito na Figura 11.11.

## 11. Casos de Uso

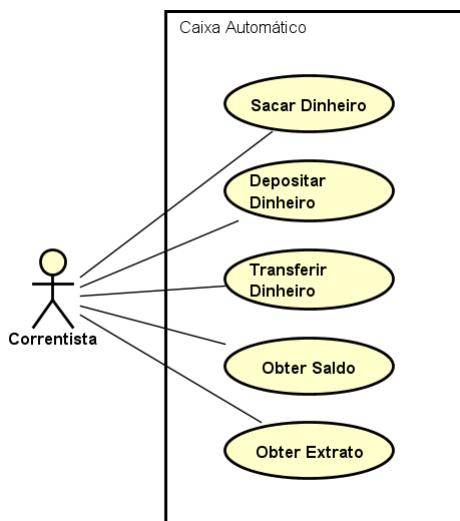


Figura 11.11.: Indicando o sistema ao qual os casos de uso se referem. Fonte:do autor.

### 11.10.2. Generalização entre Atores

Muitas vezes diferentes atores tem características comuns. Se estas características comuns puderem ser descritas como uma relação de especialização/generalização, podemos representar isso diretamente em um diagrama de caso de uso por meio da relação de generalização (usando uma seta com ponta triangular e linha cheia).

Múltiplos atores podem ter papéis comuns ao interagir com um caso de uso. A relação de generalização pode ser usada para simplificar relações entre muitos atores e um caso de uso. Ela também mostra que uma instância de um ator especializado por fazer tudo que outro tipo de ator faz.

Um exemplo típico é o da existência, em uma organização, de funcionários e gerentes. Normalmente, tudo que um funcionário pode fazer um gerente também pode fazer, mas a recíproca não é verdadeira. Assim, é possível criar vários casos de uso para o ator “funcionário” e fazer o ator “gerente” herdar de “funcionário”, adicionando-se então os casos de uso exclusivos de gerente. A Figura 11.12 mostra uma visão parcial de um sistema dedicado ao controle de funcionários, onde um ator gerente é uma especialização do ator funcionário. Dessa forma o gerente também pode fazer solicitar férias ou solicitar dispensa médica, mesmo que essa ligação não apareça diretamente no diagrama.

Também é possível usar a relação de generalização para criar atores abstratos, que representam um comportamento comum entre vários atores concretos. Um ator abstrato não existe na prática no mundo real, como o que é demonstrado no exemplo a seguir. No exemplo da Figura 11.13, 3 atores diferentes são considerados especialização de “*Profissional de Saúde*”, que é um ator abstrato, com o nome em itálico, que existe conceitualmente mesmo no mundo real, porém não existe nenhum exemplo de um profissional de saúde que não seja enfermeira, médico ou auxiliar de enfermagem.

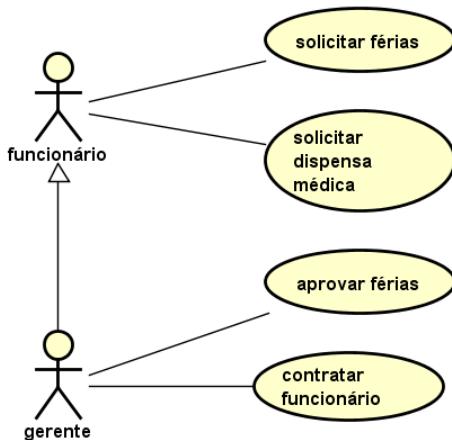


Figura 11.12.: Nesse diagrama de casos de uso, o gerente é um especialização de funcionário, logo pode fazer tudo que o funcionário faz, mais alguns casos de uso que são exclusivos dele. Fonte: do autor

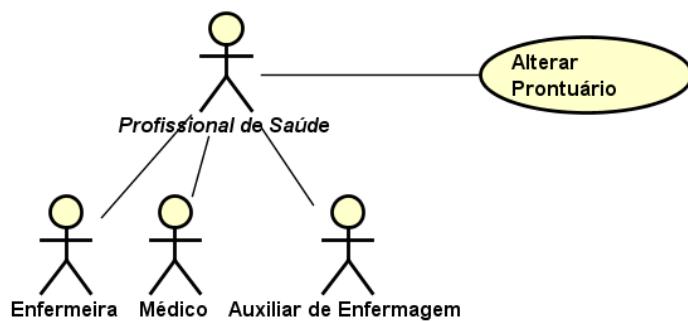


Figura 11.13.: Um *Profissional de Saúde* é a generalização de Enfermeira, Médico ou Auxiliar de Enfermagem, mas nenhum ator é um profissional de saúde se não for de um dos outros tipos. Fonte: do Autor.

### 11.10.3. Relações Entre Casos de Uso

Casos de uso podem se relacionar de 3 formas:

- pela inclusão de outro caso de uso;
- pela extensão de outro caso de uso, e
- pela generalização/especialização de outro caso de uso.

Em UML, esses relacionamentos são conhecidos como include, extend, e a generalização propriamente dita, sendo representados como na Figura 11.14.

## 11. Casos de Uso

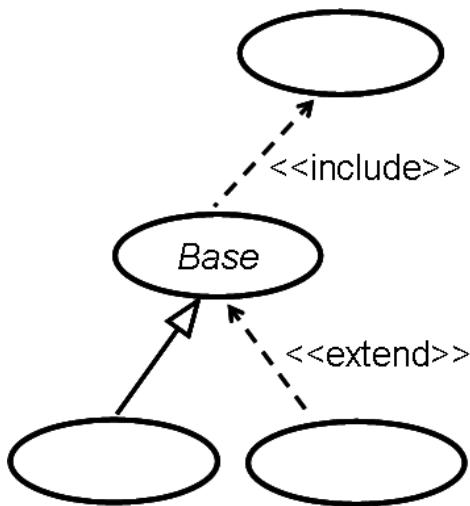


Figura 11.14.: Representações das associações entre casos de uso.

### Inclusão de Casos de Uso

O relacionamento de inclusão é o mais simples de se compreender, pois representa uma relação entre um caso de uso básico e um caso de uso incluído. Isso significa que caso de uso incluído é explicitamente inserido no caso de uso base, de forma semelhante a uma chamada de função. A Figura 11.15 apresenta um exemplo dessa associação.

O relacionamento de inclusão é usado para fatorar um comportamento comum entre dois ou mais casos de uso. Ele evita que tenhamos que descrever o mesmo comportamento duas vezes dentro dos respectivos casos de uso, aumentando a consistência e permitindo o reuso.

Também é possível usar o relacionamento de inclusão apenas para fatorar e encapsular comportamento de um caso de uso base, de forma a simplificar fluxo complexo de eventos ou remover da parte principal do caso de uso um comportamento que não é parte do objetivo primário. Nesse caso deve se considerar a complexidade do caso de uso descrito de forma completa em comparação com a complexidade de ter mais de um caso de uso sendo descrito para representar só um objetivo do usuário.

É importante entender que um caso de uso incluído é executado totalmente quando é chamado e, se não deve ser executado, a decisão é do caso de uso chamador. Alguns autores dizem que o caso de uso incluído é obrigatório, mas isso só é verdade dentro do fluxo onde ele ocorre, ou seja, um caso de uso incluído pode ocorrer em um cenário alternativo e, no caso, não ser aleatório<sup>1</sup>.

---

<sup>1</sup>Mesmo assim, alguns lugares usam como regra que casos de uso incluídos estão sempre no cenário principal e casos de uso que estendem estão sempre em um cenário alternativo. Isso é apenas uma decisão local que pode tornar o trabalho mais fácil e consistente entre várias pessoas do grupo de análise de sistema, mas não tem base no padrão UML

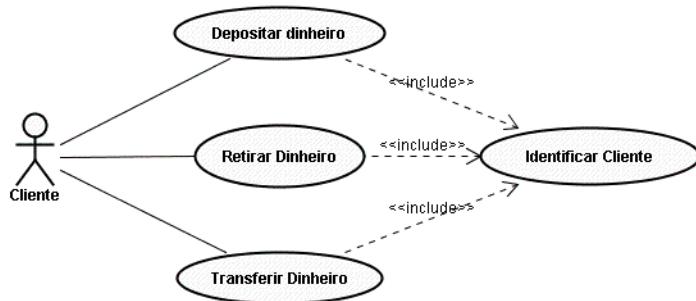


Figura 11.15.: Exemplo da associação de inclusão entre casos de uso

### Extensão de Casos de Uso

A relação de extensão descreve que um caso de uso estende o comportamento de seu caso base, o que acontece apenas se uma condição de extensão for verdade. A relação de extensão é originária do uso de *patches* em sistemas que não podiam ter seus casos de uso originais alterados para aceitar novos comportamentos e pode ser compreendida como uma forma de *patch*, do mesmo jeito que a relação de inclusão pode ser compreendida como uma chamada de função. A Figura 11.16 apresenta um exemplo de relação de extensão em um caso de uso.

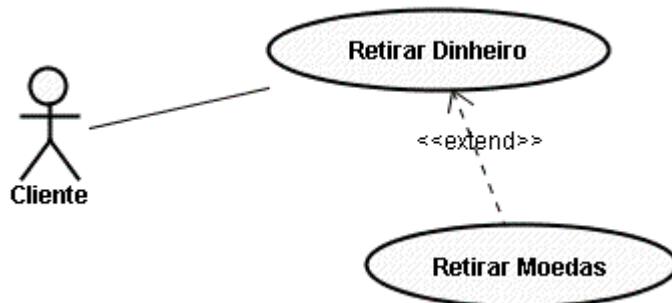


Figura 11.16.: Um exemplo da associação de extensão entre casos de uso.

Uma diferença principal entre inclusão e extensão é que a extensão sempre é chamada pelo caso de uso base, e só na extensão é que é decidido se ela deve fazer algo ou simplesmente retornar para o caso de uso base.

Extensões são tipicamente chamadas para fazer funções adicionais em cenários alternativos ou para descrever uma correção, um *patch*, no cenário principal. Por exemplo, uma possível extensão a um cenário do caso de uso “pagar com cartão de crédito”, em um terminal de ponto de venda<sup>2</sup>, seria “pagar a prazo com cartão de crédito”.

<sup>2</sup>A famosa máquina de cartão de crédito

## 11. Casos de Uso

### Generalização de Casos de Uso

O relacionamento de generalização, como estamos habituados, descreve um comportamento geral compartilhado pelo caso de uso que herda (filho) com o seu parente. Ela descreve que o “filho” tem o mesmo comportamento geral do pai, porém com alguma diferenciação (especialização).

Esse relacionamento deve ser utilizado para mostrar comportamento, estrutura ou objetivos comuns entre diferentes casos de uso; para mostrar que os casos de uso “filhos” formam uma família de casos de uso com alguma similaridade; ou para assegurar que o comportamento comum se mantém consistente. A figura a seguir mostra um exemplo de herança.

Uma instância de caso de uso executando um caso especializado vai seguir o fluxo de eventos descritos pelo caso parente, inserindo comportamento adicional e modificando seu comportamento como definido no fluxo de eventos do caso especializado.

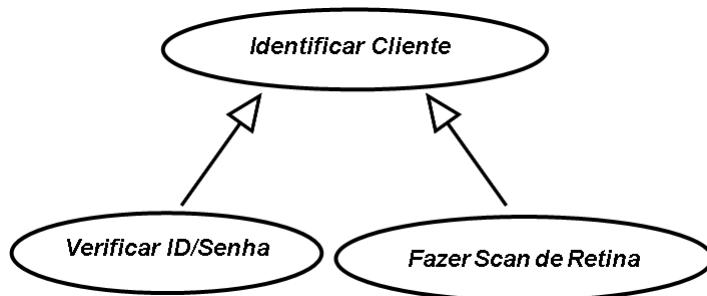


Figura 11.17.: Exemplo da associação de herança entre casos de uso

#### 11.10.4. Usando Cardinalidades em Casos de Uso

O uso mais comum de casos de uso, tendo em vista sua utilidade, é sem cardinalidades nas associações, porém é correto usá-las segundo UML.

Com o uso de indicadores de cardinalidade, o que é parte de UML, algumas informações adicionais podem ser dadas. Quando essa cardinalidade é maior que um, a especificação UML não define o que isso significa(OMG, 2017). Podem ser que os atores, múltiplos, tenham que agir em paralelo, ou em sequência, ou em outra forma.

No diagrama da Figura 11.18, existe um caso de uso apenas: aprovar empréstimo. Toda instância desse caso de uso precisa de um gerente para ser executado, porém em alguns casos especiais precisa também de um gerente regional. Gerentes e gerentes regionais, isto é, suas instâncias específicas, não precisam participar desse caso de uso. As cardinalidades marcadas indicam isso.

Também é possível que um ator execute várias instâncias de um caso de uso, ou que um caso de uso exija mais de uma instância de um ator.

## 11.11. O Que o Diagrama de Caso de Uso Não Informa

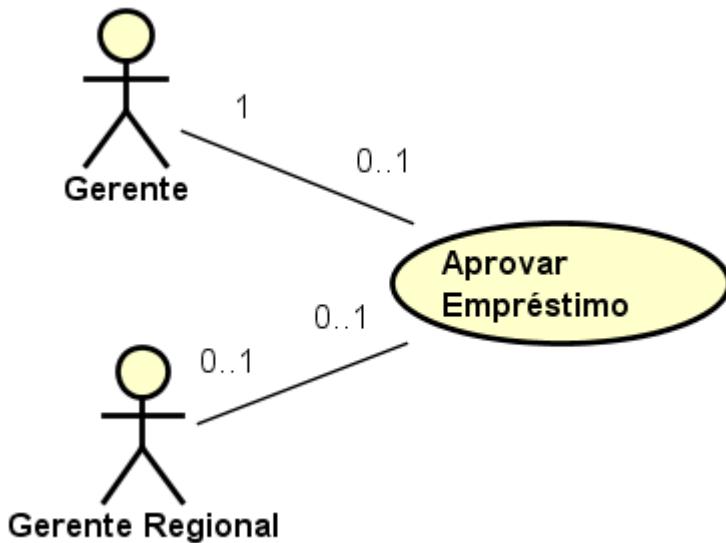


Figura 11.18.: Exemplo do uso de cardinalidades em casos de uso. Fonte: do autor

## 11.11. O Que o Diagrama de Caso de Uso Não Informa

Um diagrama de caso de uso não dá algumas informações.

Os atores principais não são informados em um diagrama de caso de uso. Mais que isso, não há nenhuma informação que ajude a entender qual a participação de um ator em um caso de uso que esteja associado a vários atores. Para facilitar o entendimento, muitos diagramas, principalmente em livros, colocam o ator principal à esquerda do caso de uso e os outros atores à direita.

Assim vendo um ou mais atores associado a um caso de uso, não é possível saber:

- se ele é o ator principal;
- se ele envia ou recebe informações para o caso de uso;
- se ele participa de todas as instâncias do caso de uso, o que pode ser resolvido em alguns casos com a indicação de cardinalidade na associação, e
- se ele está no cenário principal ou só em algum cenário alternativo.

## 11.12. Exercícios

### Exercício 11.1:

Escreva um conjunto de casos de uso para um aplicativo de celular que sirva como relógio e despertador, permitindo vários alarmes.

### Exercício 11.2:

## *11. Casos de Uso*

Vá para o site <http://jogodeanalisedesistemas.xexeo.net/> e visite a Livraria Resolve. A partir da sua visita identifique todos os atores e defina o nome de todas os casos de uso que pensa necessário. Para cada caso de uso imagine um cenário principal.

# 12

## Histórias do Usuário

Since users don't know how to solve their problems, we need to stop asking ... and to involve them instead.

(Mike Cohn)

### Conteúdo

12.1.	Conceituação de Histórias do Usuário . . . . .	199
12.2.	Um <i>Template</i> Padrão para Histórias de Usuário . . . . .	200
12.3.	INVEST . . . . .	202
12.4.	Estados de Uma História do Usuário . . . . .	206
12.5.	Exercícios . . . . .	206

#### Por que histórias do usuário?

Junto com casos de uso, histórias do usuário são uma das duas principais formas de especificar os requisitos de um software sendo usadas atualmente, principalmente em projetos ágeis.

Histórias do usuário são uma prática ágil de registro das necessidades funcionais das partes interessadas, normalmente escritas em um cartão. O nome tem certa ambiguidade, sendo usado tanto para a prática como um todo quanto para o texto principal

## 12. Histórias do Usuário

escrito no cartão. Em métodos ágeis como *eXtreme Programming (XP)* e *Scrum* elas são a principal forma de representação de um **requisito de software**,

O termo **história do usuário** foi cunhado por Kent Beck (Beck, Cockburn et al., 2014), com a intenção de descrever uma prática com maior equilíbrio entre a área de negócio e a área de desenvolvimento, sendo uma maneira mais ágil de especificar requisitos. Um dos objetivos era que, para que os usuários tivessem acesso facilitado ao método, a linguagem deveria ser bem próxima a deles, evitando as formalidades e os jargões de TI dos métodos tradicionais.

Dessa forma, histórias do usuário são uma narrativa na perspectiva do usuário (Beck, Roden et al., 2014) escrita em linguagem natural. Segundo Adzic e Evans (2014), comparando histórias do usuário com requisitos levantados unicamente por desenvolvedores ou pessoas do negócio, “histórias do usuário implicam em um modelo totalmente diferente: requisitos por colaboração”.

Inicialmente histórias do usuário tinham uma definição muito informal, e até controversa (Beck, Cockburn et al., 2014; Beck, Roden et al., 2014). O entendimento e o acordo entre os praticantes dessa metodologia foram evoluindo com o tempo e a divulgação de seu uso, por meio de artigos e livros, e acabaram sendo de certa forma padronizadas no mercado em três conceitos principais que são tratados neste texto: o *template*, a sigla *INVEST* e a regra *Card, Conversation and Confirmation*.

Beck (1999) definiu histórias do usuário, dentro do contexto de *XP* como:

- uma coisa que o cliente quer que o sistema faça;
- que possa ser feita entre uma e cinco semanas, e
- que seja testável.

Nessa definição original, o prazo de uma a cinco semanas tem mais relação com o fato da história poder ser feita junto com outras nesse tempo. Alguns autores falam das histórias isoladamente como podendo ser feitas de meio dia até 2 semanas dias (Cohn, 2004). O prazo exato, porém, não é importante, mas sim o fato que uma história do usuário que é consistente com os prazos dos métodos ágeis, e com a prática de *time-box*.

Uma outra definição, dada por Jacobson, Lawson e Ng (2019) é “Alguma coisa que o sistema de software pode ser estendido para fazer, expressa em termos do valor que vai fornecer para o usuário do sistema.”

É importante compreender, porém, que **não há nenhuma limitação formal de como são escritas**, e o importante é que sejam manipuláveis e comprehensíveis pelos usuários. Além disso, apesar da semelhança entre os nomes, **histórias do usuário não são casos de uso**, ou uma versão simplificada de casos de uso (Beck, Cockburn et al., 2014; Cohn, 2004).

Tipicamente, a parte mais visível das histórias do usuário são escritas em cartões como o apresentado na Figura 12.1.<sup>1</sup>

---

<sup>1</sup>São usados normalmente fichas pautadas de fichário

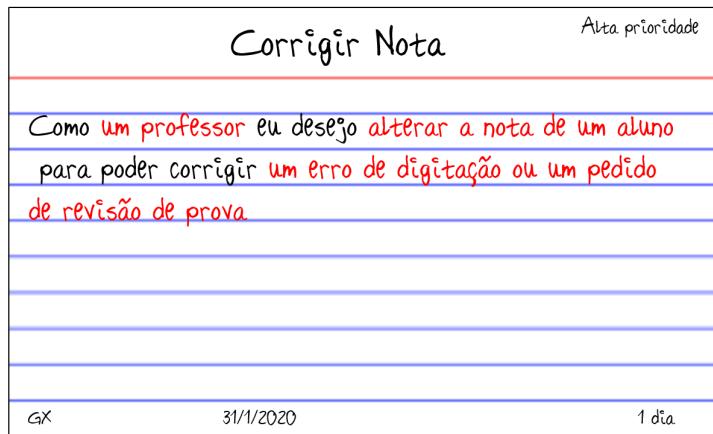


Figura 12.1.: Exemplo de uma história do usuário em um cartão. Fonte: do autor.

Como em vários métodos ágeis, é comum que os cartões fiquem afixados a um quadro, como o quadro *Kanban*, ou ainda no *Sprint Backlog* ou no *Product Backlog* do *Scrum*.

#### Por que cartões?

Vários métodos de análise de sistemas, como *Volare*, CRC e Histórias do Usuário usam cartões, ou fichas de fichário. Essa prática é comum nos EUA, mas encontra até um pouco de resistência no Brasil. A questão é que nos EUA cartões são usados em várias técnicas de estudo, como o uso de *flashcards*, ou o fichamento (que também é usado no Brasil), então o profissional normalmente já tem certo hábito com eles.

## 12.1. Conceituação de Histórias do Usuário

Uma história do usuário “descreve uma funcionalidade que terá valor” para uma parte interessada de um sistema ou software (Cohn, 2004). Ela é composta de três aspectos, conhecidos como Cartão, Conversação e Confirmação(Cohn, 2004; Jeffries, 2001):

1. o **cartão** que contém uma descrição textual da história **usada para planejamento e como um lembrete**, ou a história do usuário propriamente dita, como a que está na Figura 12.1;
2. as **conversações** entre desenvolvedores e usuários sobre a história, e que levam a descoberta de seus detalhes, e
3. as **confirmações**, as quais garantem que a história está completa e funcionando como esperado.

*Card, Conversation and Confirmation*

Segundo K. E. Wiegers e Beatty (2013, p. 146), conversações servem para refinar as histórias do usuário e também para levar dessas histórias refinadas para testes de

## 12. Histórias do Usuário

aceitação. Essas conversações podem levar ao desmembramento da história original em histórias menores e a detalhes adicionais.

Devemos deixar claro que os cartões são apenas parte de usar histórias do usuário como método, servindo como ponto focal do trabalho. A principal mudança, na verdade, é ter uma ambiente participativo, onde usuários e desenvolvedores tem encontros face a face onde se estabelecem as conversações. Possivelmente, essas conversações podem levar a alguns documentos que serão usados pelos desenvolvedores, mas não há nenhum padrão pré-concebido a seguir na documentação.

Isso causa uma pequena confusão entre o método e o artefato histórias do usuário. O método é composto pelos três aspectos, o artefato, ou a descrição da história do usuário, é registrado em um cartão. Deve ficar claro, pelo contexto, qual dos significados é tratado em cada parte do texto.

O cartão propriamente dito pode ter várias informações. A principal é a descrição da história, possivelmente na forma de um dos *templates* apresentados a seguir neste texto, ou um definido na organização.

Outra prática comum é dar um título a história. Adzic e Evans (2014) recomenda nomear as histórias bem cedo e pensar se outras partes interessadas podem estar interessadas na história, deixando os detalhes para o final.

A Figura 12.2 mostra um cartão adaptado do modelo usado originalmente na Connextra (Patton e Economy, 2014a), com suas várias partes identificadas.

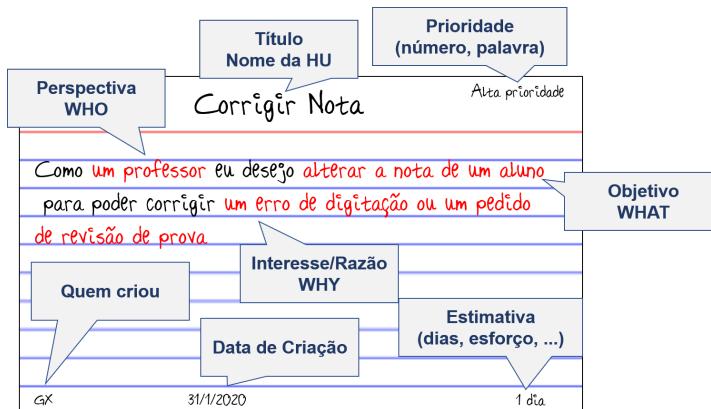


Figura 12.2.: Partes encontradas em um cartão de história do usuário, segundo um modelo original da Connextra (Patton e Economy, 2014a).

## 12.2. Um *Template* Padrão para Histórias de Usuário

North (2013) sugeriu um formato para histórias de usuário em um grupo do Yahoo, conhecido como o *template* da Connextra (Adzic e Evans, 2014), que acabou se tornando uma referência na área:

User Story  
Template

Como um *<papel de usuário>* eu desejo *<objetivo>* de forma a *<razão>*.

onde o *objetivo* descreve uma funcionalidade do sistema e a *razão* descreve porque aquela funcionalidade traz valor ao usuário. É **importante que a razão não seja apenas uma nova maneira de dizer o objetivo**, pois nesse caso o valor não está claro.

Exemplo de histórias do usuário que seguem esse padrão são:

- como um sócio do clube eu desejo marcar a quadra de futebol de forma a jogar futebol com amigos, e
- como um gerente eu desejo emitir o relatório de vendas por cliente de forma a descobrir quais são os clientes que mais gastam.

Adzic e Evans (2014) ainda recomenda que as histórias do usuário tenham duas propriedades: descrever uma mudança de comportamento, para as partes interessadas, e descrever uma mudança no sistema, para a equipe de desenvolvimento.

### 12.2.1. Outros formatos possíveis

O formato padrão responde 3 perguntas sobre o caso de uso: quem, o que e por que. É possível modificar esse formato para responder mais perguntas do 5W2H. Dois formatos possíveis são:

Como um *<papel de usuário>*, *<prazo>* , eu desejo *<objetivo>* de forma a *<razão>*.

o que responde ainda quando, como em “Como gerente, até o dia 30 de maio,eu quero o relatório de vendas por cliente ordenado por valor, porque quero conhecer meus melhores clientes.” ou ainda

Como um *<papel de usuário>*, *<prazo>* , eu desejo *<objetivo>* de forma a *<razão>*.

o que responde ainda “por quanto?”, como no exemplo “Como gerente, até o dia 30 de maio,eu quero o relatório de vendas por cliente ordenado por valor, porque quero conhecer meus melhores clientes, gastando no máximo 1 semana de desenvolvimento.”

Cohn (2004) propõe também histórias escritas em formatos mais livres, como em “Um usuário pode postar seu currículo no *website*”. Já Adzic e Evans (2014) lembra que o importante é que as histórias do usuário estimulem uma boa discussão sobre o requisito, e sugere experimentar com diferentes formatos, como perguntas, imagens.

Exemplos de histórias do usuário escritas de forma mais livre são fornecidos por Cohn (2004) para um sistema de busca de empregos:

- um candidato pode incluir seu currículo;
- um candidato pode editar seu currículo;
- um candidato pode apagar seu currículo;
- um candidato pode marcar o currículo como inativo, e

## 12. Histórias do Usuário

- um resumo pode incluir educação, empregos anteriores, publicações, salários anteriores e um objetivo.

Chamamos a atenção para a diferença entre essas histórias do usuário e como requisitos do sistema segundo a norma IEEE. Os requisitos demandam funcionalidades do sistema, no formato “o sistema deve”, enquanto as histórias do usuários dizem o que o usuário pode fazer, no formato “um usuário pode”.

### 12.3. INVEST

Ao desenvolver as histórias é necessário levar em consideração que uma boa história deve seguir alguns princípios que são definidos pelo acrônimo INVEST (Cohn, 2004; Wake, 2003):

- Independente;
- Negociável;
- com Valor;
- Estimável;
- Pequena (*Small*), e
- Testável.

#### 12.3.1. Independente

Cada história do usuário deve ser independente, sempre que possível, de outras. O benefício da independência é evitar problemas de estimativa, priorização e planejamento (Cohn, 2004). Elas devem poder ser implementadas em qualquer ordem (Wake, 2003). Isso não é sempre possível, já que algumas funcionalidades, como a produção de relatórios, precisam ser feitas uma de cada vez, e a primeira vez vai afetar, normalmente facilitando, as outras.

Também no caso de estimativas de história com alta dependência fica difícil acertar, pois a implementação de uma vai afetar a implementação da outra, mas pode não ser claro qual o impacto final. Isso pode implicar em juntar algumas histórias de modo a deixar as que as co-dependências (Cohn, 2004) estejam todas na mesma história.

Por exemplo, suponha que em um caixa automático para pagamento de estacionamento existam várias histórias, com os seguintes nomes:

- o motorista deseja **pagar com cartão de crédito** para poder sair do estacionamento com seu carro;
- o motorista deseja **pagar com cartão de débito** para poder sair do estacionamento com seu carro;
- o motorista deseja **pagar com notas** para poder sair do estacionamento com seu carro, e

- o motorista deseja **pagar com aplicativo** para poder sair do estacionamento com seu carro;

Claramente, a implementação de uma dessas histórias pode afetar as outras, tornando-as mais fáceis. A solução de juntar as histórias criaria uma só no lugar delas, usando como ação “pagar estacionamento”. Por outro lado, essa história seria grande, talvez até mesmo um épico, o que pode levar a necessitar de um outro tratamento, mais tarde no projeto.

Outra opção seria manter as histórias em separado, mas criar duas estimativas, uma para a história sendo feita isoladamente, outra para a história sendo feita depois de uma outra estar feita (Cohn, 2004).

Wake (2012) descreve 3 formas de dependência:

1. sobreposição;
2. ordem, e
3. contenção.

### **Dependência de Sobreposição**

A **sobreposição** acontece quando existem um conjunto de funcionalidades e as histórias do usuário cobrem algumas dessas funcionalidades, com interseção entre elas. Um sinal de que isto está acontecendo são histórias com a conjunção “e”, como em “Cliente compra e vende itens” e “Cliente compra e devolve itens”. Isso deveria ser dividido em histórias diferentes como nomes como (Wake, 2012):

- cliente compra itens;
- cliente vende itens, e
- cliente devolve itens

### **Dependência de Ordem**

Nesse caso há uma história que precisa ser implementada antes da outra. Os motivos podem ser muitos, inclusive ser da natureza do problema, já que um item só pode ser devolvido se for comprado. Esse tipo de dependência não pode ser eliminado, mas normalmente a prioridade do negócio vai estar na mesma ordem. ordem (Wake, 2012).

### **Dependência de Contenção**

Nesse caso existe uma organização hierárquica das histórias, onde uma contém outras. Todos os métodos tem que tratar isso de alguma forma, já que a abstração de refinamento sucessivo é muito prática e utilizada praticamente por todos para entender melhor como um sistema se organiza.

## 12. Histórias do Usuário

Histórias de usuário de nível muito alto de abstração recebem nomes como temas ou épicos (Wake, 2012) e existem formas específicas de tratá-las.

A característica principal da dependência de contenção é que a decomposição hierárquica não é uma estratégia de sequenciamento (Wake, 2012). Então é importante atacar as partes principais, seguindo normalmente uma estratégia de desenvolver o **menor produto viável** e construir versões melhores, agregando o maior valor possível, em sequência.

O **menor produto viável** (*MVP*) é a versão mais simples de um produto que pode ser lançada com uma quantidade mínima de esforço e desenvolvimento

### 12.3.2. Negociável

Uma história do usuário é na verdade um tópico de conversação entre usuário e desenvolvedores, logo elas devem ser escritas de forma a serem negociáveis. A negociação promove o entendimento e compromisso entre as partes (Jacobson, Lawson e Ng, 2019).

### 12.3.3. Com Valor

Cada história precisa ter um valor para o cliente, ou outra parte interessada que não seja da equipe de desenvolvimento (Wake, 2003), como o comprador. Histórias são sempre escritas com a perspectiva do usuário.

Esse conceito está relacionado com a prática geral dos métodos ágeis de desenvolver os sistemas por fatias verticais, isso é, sempre entregando funcionalidade ao cliente, em vez de camadas onde funcionalidades internas, de base, são feitas antes, e mais tarde as camadas externas, que atendem os clientes, são feitas.

A questão do interesse de outras partes interessadas é importante, porque usuários não estão interessados com certos requisitos do sistema. Cohn (2004) cita, por exemplo, a necessidade de seguir alguma norma internacional, que pode ser do interesse do comprador mas não dos usuários. Um história válida, então, seria “A equipe de desenvolvimento irá fornecer documentação que atende as normas de auditoria da ISO 9001”(Cohn, 2004).

### 12.3.4. Estimável

Para poder ser útil no processo de planejamento, é importante que as histórias do usuário sejam estimáveis Wake (2003) em relação ao esforço necessário para implementá-las.

Estimativas são sempre feitas pelos desenvolvedores. É preciso reafirmar a prática que as pessoas do negócio decidem o que deve ser feito, mas são os desenvolvedores que decidem como e o prazo.

Estimar histórias e dar uma avaliação do esforço necessário para desenvolvê-las, em alguma medida combinada no projeto. Dois métodos usados no mercado são usar pesos

relativos, como a sequência (1,2,3,5,8,13,20,40,100) usada no *Planning Poker* ou tamanhos de camisa, como (XP,P,M,G,XG).

Os principais motivos para histórias não poderem ser estimadas são (Cohn, 2005):

- os desenvolvedores não entenderam o domínio, isto é, não entenderam como a história do usuário deve acontecer no seu contexto, e devem discutir mais a história com os usuários;
- os desenvolvedores não sabem como implementar, por exemplo, por falta de domínio da tecnologia, e
- a história é muito grande e é muito difícil estimar o esforço necessários, sendo um épico.

É importante notar que normalmente as histórias do usuário são usadas junto com *time-boxes*, o que implica em elas terem um tamanho máximo, correspondente ao que pode ser feito dentro desse intervalo de tempo.

O *Planning Poker* é uma técnica de estimativa baseada na técnica de Delphi que usa reuniões face a face e estimativas pessoais por meio de cartas com números que seguem aproximadamente a sequência de Fibonacci

### 12.3.5. Pequena

O tamanho da história do usuário é um fator importante na sua utilidade tanto no projeto quanto no planejamento do mesmo. Histórias muito pequenas são fáceis de implementar, mas podem trazer pouco valor, enquanto histórias muito grandes trazem muito valor, mas também causam dificuldades para estimá-las ou completá-las dentro de um *time-box*. Encontrar um tamanho ideal para uma história é parte do trabalho de colaboração entre desenvolvedores e clientes. Esse tamanho deve ser pequeno de forma que ela possa ser implementada em um ciclo de desenvolvimento (Jacobson, Lawson e Ng, 2019)

Histórias grandes demais podem ser de dois tipos (Cohn, 2004):

- **histórias compostas**, que são épicos com várias histórias menores dentro dele, ou
- **histórias complexas**, que são inherentemente grandes e difícil de ser particionadas em histórias menores.

Um exemplo de história composta é a história “Um correntista pode movimentar sua conta” em um banco, que pode ser facilmente convertida em um conjunto de histórias como:

- um correntista pode depositar valor em conta;
- um correntista pode sacar dinheiro da conta;
- um correntista pode transferir valor de sua conta para outra, e
- um correntista pode pagar boleto.

Histórias grandes podem, e normalmente devem, ser quebradas. Cada história menor devem manter uma interação significativa com o usuário, mesmo que pequena ou especializada. Uma dica para escolher um pedaço de uma história do usuário durante o processo de divisão é olhar os testes e verificar seus objetivos (Jacobson, Lawson e Ng, 2019).

### 12.3.6. Testável

Esse critério é um dos mais importantes. Normalmente é usado com técnicas como *Test Driven Development*, onde o teste é implementado antes. Também ajuda a que desenvolvedor e usuário concordem com o que vai ser feito (Jacobson, Lawson e Ng, 2019).

## 12.4. Estados de Uma História do Usuário

De acordo com o SEMAT, uma história do usuário pode passar por 4 estágios (Jacobson, Lawson e Ng, 2019):

1. identificada, onde tem seu valor claramente expresso e está no *backlog* do produto;
2. pronta para desenvolvimento, onde os detalhes foram discutidos e está claro o que é necessário para cumprir seus requisitos;
3. em progresso, onde o time está desenvolvendo a história, e
4. verificada, onde um representante dos usuários verificou a história.

Já um cartão de história pode passar por 3 estados (Jacobson, Lawson e Ng, 2019):

1. valor expresso, o que pode ser feito em alguns dos formatos aceitos;
2. critério de aceitação expresso, e
3. conversação capturada, onde foram feitas as discussões que detalham os requisitos da história e elas foram possivelmente registradas no cartão ou em um sistema a parte.

Finalmente, um caso de teste passa também por 3 estados (Jacobson, Lawson e Ng, 2019):

1. critério de aceitação capturado;
2. roteiro de teste pronto, e
3. roteiro de teste automatizado.

## 12.5. Exercícios

### Exercício 12.1:

Escreva um conjunto de histórias do usuário para um aplicativo de celular que sirva como relógio e despertador, permitindo vários alarmes.

### Exercício 12.2:

Vá para o site <http://jogodeanalisedesistemas.xexeo.net/> e visite a Livraria Resolve. A partir da sua visita especifique no todas as histórias do usuário que encontrou.

## **Parte V.**

# **Desenvolvimento**



Nessa parte deve ser tratada a gestão de configuração.



# 13

## Testes

Testing shows the presence, not  
the absence of bugs

(*Edsger W. Dijkstra*)

### Conteúdo

---

13.1.	Terminologia . . . . .	212
13.2.	Introdução aos Conceitos de Teste . . . . .	212

Testes servem para mostrar empiricamente que um fragmento específico de software cumpre um conjunto dado de requisitos em um ambiente dado em um momento específico do tempo. Eles permitem verificar o comportamento do sistema, sendo uma técnica dinâmica que busca revelar falhas no software, isto é, encontrar defeitos.

Verificar significa que algo está dentro de suas especificações. Essa palavra é normalmente utilizada junto com a palavra validar, que significa que algo faz o que deveria fazer, como o usuário deseja. Diferenciar esses dois conceitos é algo importante na Engenharia de Software, pois nos mostra que mesmo fazendo o que prometemos, podemos não entregar o desejado.

Segundo Pressman e Maxim (2019), testar é “Testar é o processo de exercitar um programa com a intenção específica de achar erros antes de entregá-lo ao usuário final”. Já a ISO/IEC/IEEE (2017) define teste como “Atividade em que um sistema ou componente é executado sob condições especiais, os resultados são observados ou registrados, e uma avaliação é feita sobre algum aspecto do sistema ou componente”.

## 13.1. Terminologia

Para discutir testes, é necessário conhecer uma terminologia adequada:

- **bug**: são coisas que o software não devia fazer, ou coisas que o software não faz, mas devia fazer (Telles e Hsieh, 2001), sinônimo de falta ou defeito (ISO/IEC/IEEE, 2017).
- **crash**: a falha (*failure*) completa e repentina de um sistema computacional ou componente (ISO/IEC/IEEE, 2017).
- **defeito (defect)**: uma imperfeição ou deficiência em um produto do trabalho que não atende seus requisitos ou especificações, e precisa ser ou reparado ou trocado (ISO/IEC/IEEE, 2017).
- **erro (error)**: uma ação humana que produz um resultado incorreto, ou a diferença entre um valor ou condição computada, observada ou medida e o valor ou condição verdadeiro, especificado ou teoricamente correto, ou ainda um estado errôneo do sistema (ISO/IEC/IEEE, 2017).
- **erro (mistake)**: ação humana que produz um resultado incorreto (ISO/IEC/IEEE, 2017).
- **falha (failure)**: o fim da habilidade de um sistema de desempenhar uma função necessária ou sua inabilidade de desempenhar dentro de limites previamente especificados; um desvio externamente visível da especificação do sistema (ISO/IEC/IEEE, 2017).
- **falta ou defeito (fault)**: manifestação de um erro em software, defeito em um sistema ou uma representação de um sistema que, se executado/ativado, poderia potencialmente resultar em um erro (ISO/IEC/IEEE, 2017).

Resumindo essas definições, um erro é um problema interno e causa um defeito, que é percebido externamente. Um erro (*mistake*) é causado por um humano que gera um defeito (*fault*), que pode se transformar em um erro encontrado ou um defeito que vai gerar uma falha no ambiente do usuário.

Um processo de qualidade deve descobrir como os erros estão sendo inseridos. A qualidade do processo diminui a inserção de erros. O processo de qualidade do produto detecta os erros antes de se transformarem em defeitos, causando falhas para o usuário. A partir dessa ideia é possível definir o ciclo de vida do defeito, que se inicia com a inserção, passa pela detecção e termina quando é removido.

## 13.2. Introdução aos Conceitos de Teste

Não é possível provar que um programa está absolutamente correto. Mesmo tentativas de construir programas corretos por meio da lógica, ou de outro processo formal, falharam. O objetivo do teste é encontrar problemas, para corrigi-los.

É praticamente impossível testar um programa completamente. Para isso seria necessário testar todos os caminhos possíveis do código, todas as entradas válidas e todas as entradas inválidas. É fácil provar que isso é impossível: imagine um programa que permite a entrada de uma quantidade indeterminada de dados em um arquivo ou base de dados. Para testá-lo completamente, seria necessário entrar todas as entradas possíveis, tanto corretas quanto incorretas, em todas as ordenações possíveis. Isso certamente levaria um tempo enorme.

Dessa análise, podemos concluir que leva mais tempo para testar do que gostaríamos, e temos que fazer opções, ou planos de teste, formados de casos de teste. Partimos do princípio que nunca encontraremos o último erro, e tomamos uma postura probabilística.

Uma prática importante é a do **testes regressivos**, que significa executar todos os testes do software a cada mudança, para ter indícios que a última mudança não gerou um erro que já foi resolvido antes. Na prática acontece um ciclo da seguinte forma, quando um defeito é encontrado na operação:

1. Acontece o defeito;
2. O cliente ou usuário faz uma reclamação, que pode ser mal documentada;
3. É preciso criar um teste que replique o defeito;
4. É feita a suposição que o teste ativa o erro que provoca o defeito descrito pelo usuário (pode ser outro erro que provoca o mesmo defeito);
5. Criamos novos testes para delimitar o defeito;
6. Buscamos o erro (*mistake*);
7. Corrigimos o erro;
8. Corrigimos outros erros associados;
9. Criamos novos testes para verificar que tudo funciona, e
10. Executamos os testes.

Testes possuem uma cobertura, isso é, eles cobrem, em certa extensão, os requisitos de um sistema ou produto de software (ISO, 2017).



## **Parte VI.**

# **Métricas de Software**



# 14

## Medidas e Estimativas do Tamanho do Software

Measurement is the first step that leads to control and eventually to improvement. If you can't measure something, you can't understand it. If you can't understand it, you can't control it. If you can't control it, you can't improve it.

(H. James Harrington)

### Conteúdo

---

14.1.	Esforço . . . . .	219
14.2.	Algumas Medidas Conceitualmente Simples . . . . .	220
14.3.	Visão Geral dos Métodos de Estimativa . . . . .	224
14.4.	Métodos de Estimativa Baseado em Opinião . . . . .	225

Ao iniciar um projeto de software, ou qualquer outro projeto, é necessário ter uma estimativa da quantidade de tempo e recursos que serão usadas para desenvolvê-lo. Essa estimativa é feita a partir de outra: a estimativa de tamanho do software. Para essa estimativa ser feita é preciso, antes, entender como esse tamanho pode ser medido. Isso foi visto parcialmente no Capítulo ??, mas existe muito material sobre o assunto tanto na

## 14. Medidas e Estimativas do Tamanho do Software

área de Gestão de Projetos (PMI, 2017) quanto na área de Engenharia de Software (IEEE Computer Society, 2014a; Pressman e Maxim, 2014).

Apesar de muito estudada, a estimativa de projetos de software ainda é um problema em aberto. Isso acontece não só pela dificuldade inerente de prever alguma coisa, mas também por alguns motivos adicionais. O primeiro motivo importante é a falta de maturidade do processo de muitas organizações, que resulta na falta de dados que ajudem a fazer previsões com baixa margem de erro. Em muitos lugares ainda são praticadas estimativas *ad-hoc*, baseadas na experiência dos desenvolvedores ou gerentes, e ainda sujeitas a pressões de clientes e da alta gerência. Em outros, mesmo com processos mais definidos, não há dados suficientes sobre o desempenho da equipe em projetos anteriores e similares para fazer boas previsões. Finalmente, muitos projetos de software começam com uma definição incerta do que vai ser feito, tendo um foco inicial no problema a ser resolvido, o que dificulta também a previsão. Os métodos ágeis atacam alguns desses problemas evitando fazer grandes estimativas do trabalho total do projeto no seu início, porém isso impossibilita certas formas de contrato.

Quanto mais tradicional for o desenvolvimento, seja por cultura da empresa, seja por imposição do cliente, mais será necessário fazer e reduzir os erros dessa estimativa, inclusive por motivos contratuais. Além disso, ela influencia todas as expectativas das partes interessadas, incluindo, principalmente: o orçamento, a funcionalidade entregue e o prazo do projeto. Já nos projetos ágeis, o foco é na entrega de valor em prazos pré-delimitados e o projeto é visto de outra forma.

Logo, o objetivo da estimativa de um projeto de software exige, em um certo momento no tempo, no início ou ao longo do projeto, prever os recursos necessários, cujo custo é basicamente dominado pelos recursos humanos<sup>1</sup>, e o tempo necessário para terminar o projeto, dando ferramentas ao gestor para planejar as próximas fases. Isso é feito com um grau de incerteza, que é maior no início, por ser uma previsão baseada em informações limitadas, mas é importante lutar para que essa incerteza seja pequena e conhecida. Ao longo do projeto as estimativas continuam e são constantemente comparadas e revisadas. Como há um aumento da quantidade de informações, normalmente se tornam mais precisas. No limite, ao final do projeto, a previsão dos recursos e tempo necessário para o projeto é perfeita, apesar de ser inútil.

Este livro é basicamente dedicado a responder a pergunta **o que** devemos desenvolver, de algumas formas, mas não seria completo se não apresentasse também alguma maneira de avaliar o tamanho do software que está sendo proposto, principalmente após o processo de análise. Este capítulo e os próximos tentam mostrar como essas outras perguntas podem ser respondidas. Para isso trataremos de algumas medidas de tamanho possíveis para o software, respondendo a pergunta 2, e discutiremos como essas medidas podem ser previstas, respondendo a pergunta 3. Finalmente mostraremos, com alguma simplificação, como o método COCOMO II(Barry W. Boehm et al., 2000) pode ser usado para calcular o esforço, a equipe média e o tempo de desenvolvimento.

---

<sup>1</sup>Ou proporcional a eles

## 14.1. Esforço

Tendo em vista o forte domínio do valor dos recursos humanos em projeto de software, a principal forma de entender a demanda de recursos e tempo para um projeto de software é a quantidade de trabalho que será necessária para construí-lo. Em Engenharia de Software, o trabalho necessário para desenvolver um produto é conhecido como **esforço**. O esforço é medido em pessoas-hora ou pessoas-mês, isto é, a quantidade total de horas, ou meses, trabalhados por uma quantidade de pessoas para fazer o produto.

Assim, um software pequeno pode precisar de 4 pessoas-mês, dando a ideia que 2 pessoas podem fazê-lo em 2 meses, ou 1 pessoa pode fazê-lo em quatro meses. Porém, é difícil imaginar que 8 pessoas possam fazê-lo em 15 dias. Acontece que sistemas de informação são um pouco como bebês: não podemos ter a gestação de um bebê com nove mães em um mês, também não podemos colocar milhares de pessoas para fazer um software em um dia. Ao contrário, se colocarmos pessoas demais é possível que a necessidade de gerenciar atrapalhe a produção(Brooks, 1995). Na verdade, Barry W. Boehm et al. (2000) encontraram uma relação entre o esforço necessário e o tempo necessário para fazer um sistema, e consequentemente o tamanho médio da equipe, incluída no método COCOMO II.

A medição do esforço de um projeto em andamento ou terminado é ao mesmo tempo uma prática fácil e difícil. Em um projeto onde toda a equipe está dedicada, o esforço pode ser medido simplesmente considerando o tempo e a quantidade de pessoas envolvidas no projeto. Já em casos onde há vários colaboradores parciais, ou se deseja uma contagem mais exata, é necessário que todo o trabalho seja registrado e contado detalhadamente.

A pergunta principal do processo de estimativa de software, então, é o esforço necessário para desenvolver uma certa funcionalidade, dentro de um contexto. A expectativa de um gerente de projeto é que esse esforço seja estimado diretamente a partir de uma especificação, de preferência o mais cedo possível. O que se espera, e há uma confirmação pelo método COCOMO (Barry W. Boehm et al., 2000), é que o esforço necessário seja função, pelo menos em maior parte, do tamanho do software.

Disso podemos concluir que uma das mais importantes informações que podemos ter sobre um produto de software é seu **tamanho**. O problema é que tamanho, em software, é um conceito com muitos significados. Por exemplo, você pode simplesmente medir o tamanho de um programa na forma compilada, e isso tem impacto no processo de distribuição do produto, como tamanho de mídia ou ocupação da rede. Porém, esse tamanho tem pouco significado em relação a questões muito mais importantes, como é o esforço necessário para construí-lo. Grande parte do tamanho do programa a ser distribuído, por exemplo, pode vir de bibliotecas anexadas a ele. Qual o tamanho de um programa “Hello World!” para um sistema operacional de janelas e para um sistema de linha de comando<sup>2</sup>? Assim, entre as medidas propostas, existem algumas mais úteis do que outras, em cada contexto.

---

<sup>2</sup>Em um pequeno experimento, a diferença foi de cinco vezes

## 14. Medidas e Estimativas do Tamanho do Software

Dessa forma, temos uma sequência de questões a serem respondidas:

1. O que o software fará?
2. Como medir o tamanho de um software?
3. Como prever o tamanho do software que faz o que foi pedido?
4. Qual o **esforço** necessário para desenvolver um software desse tamanho?
5. Qual a equipe média e o tempo de desenvolvimento necessário para realizar esse **esforço**?

### 14.2. Algumas Medidas Conceitualmente Simples

Algumas medidas de software são razoavelmente simples de entender, o que permite que elas sejam usadas e compreendidas por pessoas com pouco ou nenhum treinamento, como por exemplo:

- linhas de código fonte;
- pontos de história, e
- tamanhos de camisa.

Essas medidas, apresentadas a seguir, podem ser usadas em métodos de estimativas baseados em especialistas, que serão o tema da próxima seção.

#### 14.2.1. A Medida Mais Simples: Linhas de Código

A forma mais conhecida de medir o tamanho de software é a contagem de (milhares) linhas de código fonte, conhecida como LOCs, KLOCs, SLOCS ou KSLOCs, do inglês (*Kilo*) (*Source*) Lines of Code. A ideia de contar as linhas de código do produto de software e ter uma medida direta e facilmente verificada.

Só que a questão não é tão fácil.

Primeiro, linguagens diferentes vão apresentar uma quantidade de linhas de código diferentes para a mesma funcionalidade, como mostram as Listagens 14.1 e 14.2. Como comparar o tamanho de dois programas em linguagens diferentes então?

Programa 14.1: Programa Hello World para Win32(Rösler, 1994)

```
; Hello world em Assembler para arquitetura Win32

TITLE Hello world in win32. Tasm

VERSION T310
Model use32 Flat,StdCall

start_code segment byte public 'code' use32
begin:
```

```

Call MessageBox, 0, offset sHallo, offset caption, 0
Call ExitProcess, 0
start_code Ends

start_data segment byte public 'data' use32

sHallo db 'Hello world',0
caption db "Hi",0

start_data

```

Programa 14.2: Programa Hello World em Python 3(Rösler, 1994)

```

# Hello world em Python 3
print("Hello World")

```

Segundo, o que contar como uma linha de código? Por que uma sentença com o mesmo significado pode ser escrita em um número diferente de linhas. Além disso, contam-se os comentários ou não? Contam-se os espaços em brancos destinados a facilitar a leitura? Esse problema, ao menos, foi resolvido com uma definição do que deve, e do que não deve, ser contado, por meio de padrões razoavelmente complicados. porém esse trabalho é difícil, também dependente de linguagem, e pode gerar dúvidas em função do estilo de programação.

Linhas de código tem como vantagens ser intuitivas, mesmo considerando os complicados padrões que dizem o que pode e o que não pode ser contado, e permitem a automação da contagem. A automatização acaba sendo importante na análise estatística de projetos, que levou ao método COCOMO II, por exemplo, e pode gerar adaptações locais aos valores das constantes desse método e de suas variantes(Barry W. Boehm et al., 2000).

Seus problemas, porém, são muitos:

- não têm relação com o esforço de análise e projeto, sendo ligadas apenas ao esforço de programação;
- não tem relação real com a funcionalidade entregue;
- variam de acordo com linguagem, estilo de programação, experiência do programador, uso de bibliotecas, etc.;
- não atendem as tecnologias mais modernas, com uso de GUI e geração de código;
- quando usadas como parâmetro de remuneração, incentivam soluções mais longas, e
- são difíceis de usar como medida inicial de estimativa.

Provavelmente a maneira mais difundida de contar linhas de código é a fornecida pelo Software Engineering Institute em um manual de 242 páginas, que inclui uma *check-list* de cinco páginas para exemplificar como isso pode ser feito. Para linhas de código fonte lógicas (LSC) as instruções básicas sugerem contar instruções executáveis ou não e declarações, e não contar comentários e linhas de código. Além disso, devem

## 14. Medidas e Estimativas do Tamanho do Software

ser contadas linhas programadas, geradas, convertidas, copiadas e modificadas, mas não linhas removidas (Park, 1992).

Tendo em vista o quadro de problemas da medida de linha de código, é possível pensar que uma medida realmente interessante sobre software não fala do artefato criado, mas sim do resultado fornecido pelo artefato. Assim, outras medidas podem fornecer uma ideia melhor do software.

Isso significa entender que o verdadeiro valor do software não está nas medidas diretas que podemos fazer, mas sim no que ele fornece ao usuário, e que esse tamanho que é interessante.

Essa foi a motivação da criação da medida **Pontos de Função**, que é tratada no Capítulo 15.

### 14.2.2. Pontos de História

“**Pontos de história**” é uma medida ágil usada como estimativa do esforço necessário para desenvolver uma história do usuário. Pontos de História são bastante informais e, devido a forma como são definidos e usados, não podem ser comparados entre projetos.

Segundo Cohn (2005) “pontos de histórias são uma unidade de medida para expressar o tamanho geral de uma história do usuário, funcionalidade ou outra peça de trabalho”. Além disso, Cohn (2005) avisa que “O valor bruto que damos não é importante. O que importa são os valores relativos.”

Pontos de história podem ser medidos em uma escala qualquer, porém o mercado praticamente estabeleceu o uso de um conjunto de valores que aproxima uma série de Fibonacci: 0,  $\frac{1}{2}$ , 1, 2, 3, 5, 8, 13, 20, 40 e 100. Esses números devem ser vistos como uma referência de comparação.

Essa sequência, em vez de usar qualquer número, tem uma função: evitar discussões infrutíferas e aceitar uma margem de incerteza em estimativas. Dessa forma se evita que em uma discussão para estimar o tamanho de uma história se perca tempo discutindo se é 12 ou 13, por exemplo. Os números dessa série, então, tem a propriedade de mostrarem não só tamanhos claramente distintos, mas também com uma importância nessa distinção: cada número é, aproximadamente, a soma dos dois anteriores, assim a comparação entre tamanho histórias pode ficar mais clara.

Para fazer a estimativa, em cada início de projeto, a história do usuário mais simples, para aquele projeto específico, recebe o valor 2 como estimativa do seu esforço. A partir daí, os outros valores são usados no processo de estimativa escolhido, tentando mostrar a relação proporcional de esforço necessário em uma tarefa em relação a tarefa de peso 2. Claro que nenhuma história no início do projeto vai ser pontuada com um valor menor que 2, mas é comum aparecerem histórias menores com o tempo.

A forma normalmente escolhida para fazer a estimativa é o *Planning Poker* (Cohn, 2005, p 56), principalmente em projetos ágeis.

A estimativa de pontos de história é calibrada ao longo do projeto, pois a equipe vai aprendendo a estimar melhor ao realizar as histórias.

Alguns autores, como ISBS (2010), propõem que a equipe analise a relação entre os pontos de histórias atribuídos a uma história de usuário e as horas efetivamente gastas para implementá-las. Isso pode trazer um visão mais clara aos membros da equipe, porém não é necessariamente vantajoso frente a experiência real da equipe no projeto e depende, para ser uma ação prática, do método de gestão levantar essas informações com facilidade, como acontece com o uso de algumas ferramentas de gestão de projetos ágeis.

### Velocidade da Equipe

A velocidade de uma equipe ágil é o número de pontos de história que ela faz, em média, em cada ciclo (ou *sprint*). No início do projeto, como a equipe não sabe ainda estimar muito bem os tamanhos das histórias, essa velocidade tende a variar. Ao longo do projeto ela costuma se tornar estável e até mesmo diminuir com o aumento da produtividade da equipe. Vigiar a velocidade é uma tarefa dos gestores ágeis, porém ela é apenas um sinalizador, não é razoável utilizá-la como um número absoluto de produtividade, porque as equipes vão adaptando suas estimativas ao longo do projeto.

### 14.2.3. Tamanhos de Camisa

Uma medida curiosa relatada por Dimitrov (2020), Rubin (2013) e vários outros autores é medir o tamanho de software fazendo uma analogia com tamanhos de camisa. Nesse caso, usado em métodos ágeis, cada história do usuário recebe um valor entre os tamanhos que encontramos para camisetas. Uma seleção possível de valores seria: XP, P, M, G, XG, XXG.

*T-Shirt Sizing*

Para isso funcionar, é possível manter uma tabela que associe cada tamanho com uma margem de variação, possivelmente em horas de trabalho, para estimativas de artefatos pequenos, mas talvez de custo se são estimados softwares completos, por exemplo.

Essa escala estimula a comparação e a abstração, porém dificulta a determinação de valores como a velocidade da equipe e o tamanho global de um projeto ou de uma fase de projeto.

Rubin (2013) considera que medidas de tamanho de camisa associadas a faixas de valores de custo são boas para o planejamento de portfolio.

### 14.3. Visão Geral dos Métodos de Estimativa

Como visto, os motivos principais para se medir software têm relação com a tarefas de gerenciar projetos de software. Precisamos saber o tamanho para entender quanto trabalho fizemos ou vamos fazer, qual a dificuldade de fazê-lo, ou qual será o custo. Entender o tamanho de um produto permite responder perguntas como:

- Qual o custo de desenvolver o sistema?
- Qual o esforço para desenvolver o sistema?
- Quantas pessoas serão necessárias para fazer esse software?
- Em quanto tempo o sistema ficará pronto?
- Que recursos são necessários para o sistema executar?
- Qual o valor da nossa carteira de software?

Métodos de estimativa de projeto de software podem ser divididos em três grandes grupos (ISBS, 2010):

- métodos macro, que buscam estimativas de alto nível;
  - uso de equações;
  - comparação, com projetos similares, e
  - analogia, que busca um projeto completo muito similar;
- métodos micro, que buscam estimativas a partir dos detalhes, baseados na estrutura analítica do projeto ou em outra forma de quebrar sucessivamente o projeto em tarefas menores que podem ser estimadas com mais facilidade;
- métodos usando IA, que não serão tratadas neste texto.

Estimar software sem o uso de IA sempre implica em ter, em algum momento, a opinião de pessoas. Algumas práticas se baseiam fortemente nas opinião de especialistas, talvez um único especialista. Outros, principalmente os aplicados em processos ágeis, se utilizam da equipe e tendem a cultivar o aprendizado e praticar revisões nas estimativas.

A questão aqui é realmente ligada ao tipo de contrato de desenvolvimento que está sendo feito. Um contrato tradicional não tem como escapar de fazer uma estimativa mais formal, considerar riscos e cobrar valores que vão cobrir o custo e gerar lucro. Um contrato por tempo de serviço, onde a funcionalidade é determinada de tempos em tempos pelo cliente, facilita o processo de estimar, principalmente se este intervalo de tempo for curto.

É importante deixar claro que o problema de estimar software não é complicado para pequenas tarefas, mas extremamente difícil para grandes sistemas. Mesmo dividindo um software com uma metodologia como a Estrutura Analítica de Projeto, há sempre o problema de integrar as partes e a complexidade do sistema não é o somatório das complexidades das partes. Pelo contrário, quanto maior o sistema, mais complexo é construí-lo. Essa afirmação é corroborada pelos resultados da pesquisa que levou ao método COCOMO II (Barry W. Boehm et al., 2000).

## 14.4. Métodos de Estimativa Baseado em Opinião

Alguns métodos simples para estimar baseado em opiniões são:

- Método de Estimativa do PERT;
- Método de Delphi, e
- *Planning Poker*.

### 14.4.1. O Método de Estimativa do PERT

Uma maneira simples simples de fazer uma estimativa através da opinião de especialistas é a usada pelo método PERT (ISBS, 2010; PMI, 2017), que estima um valor a partir da fórmula:

$$T_e = \frac{T_o + 4 \times T_\mu + T_p}{6} \quad (14.1)$$

onde  $T_p$  é a estimativa mais pessimista,  $T_o$  a estimativa mais otimista e  $T_\mu$  a estimativa média. Essas três estimativas são idealmente calculadas a partir de várias opiniões dadas por pessoas diferentes, mas podem ser até mesmo fornecidas por um especialista único.

### 14.4.2. O Método de Delphi

Outra forma conhecida de fazer estimativas é o método de Delphi (Dalkey, 1967; Helmer-Hirschberg, 1967; ISBS, 2010), cuja prática original é um ciclo iterativo onde:

1. especialistas recebem uma ou mais perguntas para fazer sua estimativa;
2. as estimativas são coletadas e relatadas a todos os especialistas de forma anônima;
3. os especialistas podem revisar e dar justificativas para suas novas estimativas, e
4. o ciclo é repetido até as opiniões convergirem.

Esse método, proposto em 1966, recebe o nome do Oráculo de Delphi, ou Delfos, e vem sendo reconstruído de várias formas (ISBS, 2010), como em reuniões onde não há anonimato e em aplicativos na rede, porém a que tem mais feito sucesso nos últimos anos é o *Planning Poker*.

O Oráculo de Delfos foi um templo na Grécia Antiga onde sacerdotisas faziam previsões do futuro, normalmente ambíguas.

### 14.4.3. O Planning Poker

No *Planning Poker* (Cohn, 2005, p 56) cada membro da equipe deve fazer uma estimativa do esforço necessário para desenvolver um história do usuário. Geralmente esse esforço é medido em valores arbitrários, como os **pontos de história**, descritos na seção 14.2.2.

Para o *Planning Poker* é necessário um baralho especial, ou um aplicativo que simule esse baralho, e que possua a sequência de pontos de história usada pela sua equipe e algumas cartas adicionais, como uma carta com uma interrogação, que indica que

## 14. Medidas e Estimativas do Tamanho do Software

o estimador está sem ideia de como estimar, e uma xícara de café, que indica que o estimador está cansado e precisa parar um pouco.

As rodadas, inspiradas no Método de Delphi, são iniciadas com os membros da equipe ouvindo e questionando um *product owner* ou um analista sobre como deve ser o comportamento da história do usuário sendo estimada (Figura 14.1a). A cada rodada, cada membro do time escolhe uma carta em segredo com um número desejado(Figura 14.1b). As cartas são todas mostradas simultaneamente (Figura 14.1c) e, então, se as estimativas forem diferentes, o mais otimista e o mais pessimista devem justificar seu voto (Figura 14.1d)e o processo é refeito até que uma condição de parada seja aceita (Figura 14.1e e 14.1f).

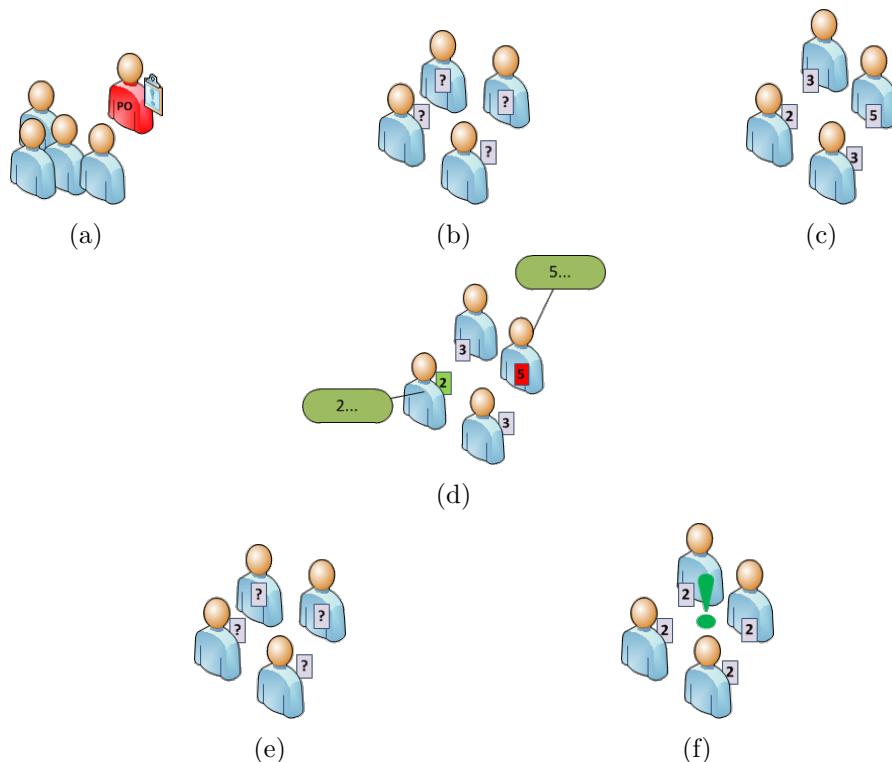


Figura 14.1.: Ilustração da sequência do Planning Poker com duas rodadas

A questão da condição de parada é um pouco controversa. Normalmente o número de rodadas é limitado a 3, com a expectativa que **um número comum a todos seja alcançado**. Caso isso não aconteça, existem várias regras de como agir: tentar mais algumas vezes, adiar a avaliação para outra ocasião, propor a quebra da história em histórias menores, usar o valor mais frequente(Cohn, 2005), usar a previsão mais pessimista(Maximini, 2018), usar a previsão mais otimista(ISBS, 2010), ou até mesmo gerar uma solução *ad-hoc* de consenso naquele instante. Essa decisão também depende da variação das estimativas. Um caso onde todas são iguais menos uma é diferente de um caso onde as estimativas estão espalhadas entre 3 números.

# 15

## Análise de Pontos de Função

### Conteúdo

---

15.1.	Visão Geral de Pontos de Função . . . . .	228
15.2.	Procedimento de Contagem . . . . .	228
15.3.	Funções Transacionais . . . . .	229
15.4.	Funções de Dados . . . . .	230
15.5.	Identificando Funções de Negócio . . . . .	230
15.6.	Identificando Entradas . . . . .	231
15.7.	Identificando Arquivos Internos . . . . .	233
15.8.	As Perguntas . . . . .	236
15.9.	Cálculo dos Pfs Finais . . . . .	236
15.10.	Conclusão . . . . .	237

### Por que Pontos de Função?

Ao fazer uma modelagem de sistema, é importante ter alguma ideia do esforço necessário para desenvolvê-lo. A Análise de Pontos de Função é uma prática aceita em todo mundo que permite avaliar o tamanho previsto de um software.

Até 1979 não tínhamos uma boa medida do tamanho do software em função da sua funcionalidade como vista pelo usuário. Apenas nesse ano, (Albrecht, 1979) apresentou uma medida conhecida como Pontos de Função, cujo objetivo era servir como avaliador e preditor do tamanho de um sistema. Atualmente Pontos de Função são uma medida

## 15. Análise de Pontos de Função

internacionalmente aceita e uma das formas possíveis de se fazer um contrato com o governo no Brasil (Brasil, 2018)<sup>1</sup>.

### 15.1. Visão Geral de Pontos de Função

Um Ponto de Função (PF) é uma medida abstrata e relativa que conta “o número de funções de negócio entregues ao usuário”. Um relatório simples, por exemplo, pode medir “4 Pontos de Função”. Da mesma forma que um metro ou um litro, Pontos de Função só fazem sentido quando comparados com um padrão. Assim, um sistema com 1.000 PF entrega o dobro de funcionalidade de que um sistema com 500 PF. Com o tempo, aprendemos a ter uma ideia absoluta do tamanho de um sistema a partir da contagem de seus PFs (IFPUG, 2012).

Pontos de Função avaliam o tamanho do software a partir de seis características principais (IFPUG, 2012):

- entradas;
- saídas;
- consultas;
- arquivos lógicos internos;
- interfaces lógicas externas, e
- ajustes de complexidade.

A maneira padronizada de contar pontos de função é fornecida pelo *International Function Points User Group (IFPUG)*(IFPUG, 2012), que fornece aos seus associados um manual contendo detalhes do que deve e do que não deve ser contado. Esse capítulo é apenas uma introdução ao método, servindo para fazer cálculos aproximados do número de pontos de função.

### 15.2. Procedimento de Contagem

O procedimento genérico de contagem de Pontos de Função é representado na Figura 15.1.

Ele se inicia com a determinação do **propósito da contagem**, isto é, a explicitação do motivo da contagem estar sendo realizada. Normalmente esse propósito estará relacionado a fornecer uma resposta a um problema de negócio existente, como a contratação de um serviço.

A partir do propósito podemos determinar o tipo de contagem. São considerados três tipos de contagem:

---

<sup>1</sup>A especificação brasileira de Pontos de Função para o governo é mais complexa na sua forma de usar, mas parte dos mesmos princípios.

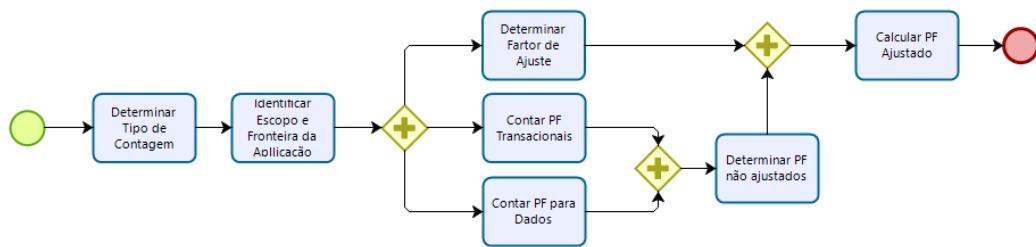


Figura 15.1.: O processo geral de contagem de Pontos de Função. Fonte: do autor

1. **projeto de desenvolvimento**, onde medimos as funcionalidades entregues ao usuário em uma versão onde o software é desenvolvido desde o início. Inclui as funcionalidades necessárias para a conversão de dados, mesmo que usadas uma única vez;
2. **projeto de melhoria**, onde medimos as modificações que alteram, adicionam ou removem funcionalidades em uma aplicação existente. Inclui as funcionalidades necessárias para a conversão de dados, mesmo que usadas uma única vez, e
3. **aplicação**, onde medimos uma aplicação já instalada. Também é chamada de *baseline* e é normalmente realizada no fim de um projeto de desenvolvimento e mantida atualizada nos projetos de melhoria.

O escopo da contagem define um conjunto ou um subconjunto do sistema, e permite dizer se uma funcionalidade deve ou não ser contada.

A fronteira da aplicação delimita o software medido, definindo sua interface com o mundo exterior. Ela servirá não só para considerarmos se uma função deve ou não ser contada, mas também para considerar se um arquivo lógico deve ser contado como interno ou externo a aplicação.

Definido o tipo de contagem, a fronteira e o escopo da contagem, que influenciarão a forma final de contagem, identificamos as funções de negócio como percebidas pelo usuário, dividindo-as em 5 grupos, agrupados em 2 tipos: funções transacionais e funções de dados.

## 15.3. Funções Transacionais

Um processo elementar é a menor atividade significativa para o usuário de forma indivisível. Isso significa que o usuário vê como um processo único, completo e independente de outros, que se inicia com o sistema em um estado consistente e termina com o sistema em um estado consistente.

Um processo elementar será contado como uma de três possíveis formas de função transacional:

## 15. Análise de Pontos de Função

- **saídas externas** (SE ou EO), são informações de negócio que o usuário final pode receber, representando relatórios, telas e mensagens de erro como um todo e não em suas partes individuais;
- **consultas externas** (CE ou EQ), que são saídas simples e imediatas, sem alteração na base, usualmente caracterizáveis por chaves simples de consulta, e
- **entradas externas** (EE ou EI), que são processos elementares que processam informações de negócio recebidas pelo sistema de fora da fronteira da aplicação e cuja finalidade principal é manter um Arquivo Lógico Interno .

### 15.4. Funções de Dados

As funções de dados estão divididas em dois tipos:

- **arquivos lógicos internos** (ALI ou ILF), que contém os dados permanentes, relevantes para o usuário e mantidos e utilizados pelo sistema. O sistema cria, altera e apaga esses dados, e
- **arquivos de interface externos** (AIE ou EIF), que contém dados permanentes e relevantes para os usuários, guardados em algum lugar por outra aplicação, mas referenciados pela aplicação em questão, sendo que outro sistema mantém esses dados.

O segundo passo é determinar a complexidade de cada função de negócio. A complexidade fornece um peso para cada função de negócio encontrada. O somatório do número de funções multiplicadas pelo seu peso fornece o número básico de pontos de função. Esse número é um indicador preliminar do tamanho do sistema.

Finalmente, no terceiro passo, o número básico de pontos de funções é corrigido em função de fatores que aumentam ou diminuem a complexidade do sistema.

### 15.5. Identificando Funções de Negócio

Para identificar as funções de negócio devemos partir de algum documento que aponte as funções **aprovadas** e pelo usuário e **úteis** para o negócio. Não devem ser contadas funções necessárias por causa da tecnologia aplicada. Basicamente, só é cobrado o que o usuário pode ver e está disposto a pagar. Também é importante que as funções de negócio sejam cobradas **como o usuário as percebem**. Isto significa que não interessa se estamos usando um ou vinte arquivos para guardar uma informação, mas sim de quantas formas o usuário pode acessar essa informação.

Além disso, devemos identificar as funções seguindo certa ordem. A ordem é importante porque encontrar um tipo de função de negócio ajuda a encontrar as funções de outro tipo. Assim, em um sistema novo devemos usar a ordem: saídas, consultas, entradas,

arquivos e interfaces. Por outro lado, em um sistema já existente devemos usar a ordem arquivos, interfaces, saídas, consultas e entradas.

Para serem contados as funções devem:

- beneficiar claramente o usuário;
- ser especificamente aprovadas pelo usuário, e
- influenciar em algum grau mensurável o projeto, desenvolvimento, implementação e suporte à aplicação.

No manual da IFPUG podemos encontrar vários exemplos que nos permitem dirimir as dúvidas de como realizar a contagem. A técnica é simples, porém difícil de dominar.

### 15.5.1. Identificando Saídas

Para contar as saídas é necessário contar todas as informações que o sistema gera para o ambiente de forma procedural. Tipicamente, saídas são relatórios impressos, telas, janelas e arquivos gerados para outras aplicações.

Devem ser contadas como saídas distintas cada formato utilizado. Basicamente, se for necessário fazer outro procedimento para produzir a informação, contamos como uma saída distinta. Também contamos cada tipo de lógica utilizada para fazer gerar a informação. Assim, se um relatório de vendas contém as vendas por vendedor e por loja, contaremos como duas saídas, pois são necessários procedimentos lógicos distintos para calcular cada um desses valores. Linhas de sumário e total, porém, não são contadas como novas saídas.

Uma saída externa pode ter uma “parte de entrada”, para, por exemplo, selecionar os registros necessários em um relatório, usando alguns campos como filtro. Essa “parte entrada” não é contada a parte, já está considerada nessa contagem.

### 15.5.2. Identificando Consultas

Consultas são, na prática, saídas simplificadas. Normalmente utilizadas para achar informações para modificá-las ou apagá-las. São sempre no monitor, não existe uma consulta em relatório de papel. Uma consulta não pode calcular nenhum valor. Em caso de cálculo de qualquer valor, temos uma saída.

## 15.6. Identificando Entradas

Entradas permitem adicionar, modificar e apagar informações. Se uma tela permite estas 3 funções, são contadas 3 entradas. Normalmente as funções de modificar e apagar ainda exigem consultas correspondentes para achar a informação que será alterada.

## *15. Análise de Pontos de Função*

Um comando específico para o sistema executar algo é uma entrada.

Mensagens de erro que fazem parte de um processo não são contadas isoladamente, mas sim como um DET. Por exemplo, se você esquecer de colocar um campo obrigatório e receber uma mensagem de erro, não deve contar uma saída a mais, e sim essa mensagem como um DET da entrada ou consulta.

Mensagens de notificação, por outro lado, são processos elementares e devem ser contados como uma saída a parte. Por exemplo, ao tentar comprar um produto que não existe e receber uma mensagem com essa notificação foi feito um processamento, que é contado como uma saída externa adicional.

### **15.6.1. Entendendo as Lógicas de Processamento**

A lógicas de processamento, ou formas de processamento lógico, são as características que uma transação tem de acordo com os requisitos solicitados especificamente pelo usuário.

A partir dessas lógicas de processamento podemos determinar, como indicado na Tabela 34, qual o tipo de função transacional que deve ser contado.

Tabela 15.1.: Tabela de apoio a determinação do tipo de função transacional.  
 P=possível, N=Não permitida, O=obrigatório, O<sup>\*</sup>= obrigatório que pelo menos uma das condições aconteça.

Forma de Processamento Lógico	Tipo de Função Transacional		
	EE	SE	CE
Realiza validação dos dados	P	P	P
Realiza cálculos ou fórmulas matemáticas	P	O <sup>*</sup>	N
Converte valores equivalentes	P	P	P
Filtrar dados e seleciona usando critérios específicos para comparar múltiplos conjuntos de dados	P	P	P
Analisa condições para determinar qual é aplicável	P	P	P
Altera ou inclui ao menos um ILF	O <sup>*</sup>	O <sup>*</sup>	N
Referencia ao menos um ILF ou EIF	P	P	O
Recupera dados ou informações de controle	P	P	O
Cria dados derivados	P	O <sup>*</sup>	N
Altera o comportamento do sistema	O <sup>*</sup>	O <sup>*</sup>	N
Prepara e apresenta informações fora das fronteiras do sistema	P	O	O
É capaz de aceitar dados ou informação de controle que entra pela fronteira da aplicação	O	P	P
Reordena ou reorganiza um conjunto de dados	P	P	P

## 15.7. Identificando Arquivos Internos

Arquivos representam um agrupamento lógico requerido pelo usuário. Podem incluir uma ou mais tabelas ou entidades.

Esse é uma das partes mais difíceis da contagem de pontos de função, pois devemos separar o que o usuário pensa do modelo que criamos. Nossa modelo muitas vezes usa vários grupos de dados, ou tabelas, ou entidades, para modelar algo que o usuário vê como um conceito único. Mesmo na modelagem conceitual, a tendência do analista é incluir entidades que o usuário não “vê” naturalmente.

Um Tipo de Elemento de Registro (RET, Record Element Type) é um subgrupo de elementos de dados dentro de um arquivo ou interface. Na prática, em um modelo de dados, um arquivo do usuário (ILF) é composto de um ou mais objetos do modelo.

Outra característica difícil de contar é que cada forma de acesso a um arquivo lógico conta novamente. Assim, por exemplo, se o usuário exige acessar um automóvel tanto por sua placa quanto por seu número do chassi, temos 2 arquivos lógicos para contar.

## 15. Análise de Pontos de Função

Exemplos: Uma nota fiscal é um arquivo lógico, com dois RETs: dados da nota, item de nota.

### 15.7.1. Identificando Arquivos Externos

Arquivos Externos representam um agrupamento lógico requerido pelo usuário. Podem incluir uma ou mais tabelas ou entidades.

Arquivos Externos são mantidos por outras aplicações. Arquivos importados contam também como Entrada Externa, arquivos exportados contam também como Consulta Externa ou Saída Externa.

### 15.7.2. Identificando Itens de Dados (DETs)

Itens de dados ou elementos de dados (DETs) são campos únicos, reconhecidos pelos usuários, desconsiderando-se recursão e repetição. DETs também podem invocar ações.

Exemplos de DETs em entradas externas são:

- campos de entrada de dados;
- mensagens de erro;
- valores calculados que são guardados;
- botões;
- mensagens de confirmação, e
- campos repetidos contam apenas como um DET.

Exemplos de DETs em saídas externas são:

- campos em relatório;
- valores calculados
- mensagens de erro;
- cabeçalhos de coluna que são gerados, e dinamicamente em um relatório

Exemplos de DETs em consultas externas (além dos anteriores que se enquadrem) são:

- campos usados em filtros de procura, e
- o clique do mouse.

Em GUIs, botões onde só se pode fazer uma seleção entre muitas (normalmente radio buttons) devem ser contados como um DET apenas. Já *check boxes* são normalmente contadas uma a uma. Botões de comando devem ser contados como um elemento de dados levando em conta o fato de executarem uma função.

## 15.7. Identificando Arquivos Internos

Tabela 15.2.: Cálculo da complexidade das saídas externas.

Arquivos Referenciados	Itens de dados referenciados		
	1-5	6-19	20+
<b>0-1</b>	Simples(4)	Simples(4)	Média(5)
<b>2-3</b>	Simples(4)	Média(5)	Complexa(7)
<b>4+</b>	Média(5)	Complexa(7)	Complexa(7)

Tabela 15.3.: Cálculo da complexidade das entradas externas.

Arquivos Referenciados	Itens de dados referenciados		
	1-5	6-19	20+
<b>0-1</b>	Simples(3)	Simples(3)	Média(4)
<b>2</b>	Simples(3)	Média(4)	Complexa(6)
<b>3+</b>	Média(4)	Complexa(6)	Complexa(6)

Tabela 15.4.: Cálculo da complexidade das consultas externas.

Arquivos Referenciados	Itens de dados referenciados		
	1-5	6-19	20+
<b>0-1</b>	Simples(3)	Simples(3)	Média(4)
<b>2-3</b>	Simples(3)	Média(4)	Complexa(6)
<b>4+</b>	Média(4)	Complexa(6)	Complexa(6)

Tabela 15.5.: Cálculo da complexidade dos arquivos lógicos internos.

RETs	Itens de dados referenciados		
	1-19	20-50	51+
<b>1</b>	Simples(7)	Simples(7)	Média(10)
<b>2-5</b>	Simples(7)	Média(10)	Complexa(15)
<b>6+</b>	Média(10)	Complexa(15)	Complexa(15)

Tabela 15.6.: Cálculo da complexidade das interfaces lógicas externas.

RETs	Itens de dados referenciados		
	1-19	20-50	51+
<b>1</b>	Simples(5)	Simples(5)	Média(7)
<b>2-5</b>	Simples(5)	Média(7)	Complexa(10)
<b>6+</b>	Média(7)	Complexa(10)	Complexa(10)

## 15.8. As Perguntas

São 14 as perguntas que devem ser feitas e ajudaram a determinar a quantidade de PF relativa a um sistema. Cada uma deve ser respondida com um número, de 0 a 5, indicando a importância da característica que se pergunta sobre o sistema, da seguinte forma:

- 0 - Não tem influência
- 1 - Influência incidental
- 2 - Influência moderada
- 3 - Influência média
- 4 - Influência significativa
- 5 - Influência essencial em todo o sistema

Para cada pergunta, o padrão IFPUG determina tipos de respostas padronizadas que nos permitem dar a resposta (entre 0 e 5) mais facilmente, como é exemplificado no item 1 (Comunicação de Dados). Foge ao objetivo desse texto fornecer um detalhamento completo do padrão de contagem, que pode ser obtido em (IFPUG, 2010).

- Quantas facilidades de comunicação existem para facilitar a transferência ou troca de informação com a aplicação ou sistema?
- Como será tratada a distribuição de dados e processamento?
- O usuário exige tempos de resposta ou throughput, ou seja, o desempenho é crítico?
- Quão fortemente é utilizada a plataforma (hardware) onde a aplicação vai ser executada?
- Qual a frequência das transações (diárias, semanais, altas o suficiente para exigir um estudo de desempenho)?
- Que percentagem das informações é inserida on-line? Se mais de 30
- A aplicação é projetada para eficiência para o usuário final?
- Quantas ILFs são alteradas por transações on-line?
- A aplicação tem processamento lógico ou matemático extensivo?
- A aplicação é desenvolvida para atender um ou muitos tipos de usuários diferentes?
- Qual a dificuldade de conversão e instalação?
- Qual a eficiência e grau de automação de inicialização, backup e recuperação?
- A aplicação foi especialmente projetada, desenvolvida e suportada para funcionar em locais diferentes em diferentes organizações?
- A aplicação foi especialmente projetada, desenvolvida e suportada para facilitar mudanças?

## 15.9. Cálculo dos Pfs Finais

Os PF são calculados em etapas: contam-se os números de entradas, saídas, consultas, arquivos e interfaces do sistema; Para cada entrada se determina um grau de complexidade,

<b>Função</b>	<b>Contagem Total</b>	<b>Complexidade</b>						<b>Total</b>
		<b>Simples</b>	<b>x</b>	<b>Média</b>	<b>x</b>	<b>Complexa</b>	<b>x</b>	
Saídas Externas		4		5		7	=	
Consultas Externas		3		4		6	=	
Entradas Externas		3		4		6	=	
Arquivos Lógicos Internos		7		10		15	=	
Arquivos de Interface Externos		5		7		10	=	
<b>Total</b>								

Figura 15.2.: Planilha de registro para ajudar a contar os Pontos de Função

de acordo com as tabelas complexidade da função , e somando os resultados (Tabela 40) se obtém a contagem básica de pontos de função; responde-se a uma série de perguntas, as quais fornecem, cada uma, um valor de 0 a 5 ( $p_i, 0 \leq i \leq 5$ ), calcula-se o número de pontos de função com a equação:

$$PF = \text{total-de-2} \times (0,65 + 0,01 \times \sum(p_i)). \quad (15.1)$$

Devemos notar que se o cálculo de PF for usado para fazer previsões seguindo o método COCOMO, apenas a contagem básica é necessária (só devem ser feitos os passos 1 e 2).

## 15.10. Conclusão

Este capítulo é apenas uma introdução aos Pontos de Função. Este método é bastante padronizado e tem muitas dicas de como contar, mas para nossa necessidade, ter uma ideia do tamanho do sistema, a forma de contagem aqui apresentada é razoável. Não foram feitas simplificações no método básico, mas nos padrões são tratados detalhes que podem levar a diferenças na contagem entre um profissional e a realizada por quem usou esse capítulo. Isso não é muito grave, já que o autor conhece casos onde empresas diferentes contam Pontos de Função de um mesmo sistema de forma diferente, com disparidades de 100%.

O melhor também é que seja feita apenas a contagem básica, por alguns motivos. O primeiro é que ela é mais aceita no mundo todo, sendo que alguns países só normatizaram ela. O segundo é que, para dar as respostas as perguntas existem definições bem mais precisas no padrão da IFPUG, e só demos uma leitura geral. O terceiro é que algumas dessas perguntas são muito antigas em relação a tecnologia.



# 16

## Uma Visão Reduzida do Modelo COCOMO II

### Conteúdo

---

16.1.	Dois Modelos em Um . . . . .	240
16.2.	Esforço em Função do Tamanho . . . . .	240
16.3.	Existe uma Relação Entre Esforço e Tempo . . . . .	241

#### Por que COCOMO II?

Conhecer as relações entre o tamanho do projeto e o esforço necessário para desenvolvê-lo, e ainda o tempo necessário para esse esforço, é um passo importante na finalização da análise de sistemas e ainda para permitir escolhas de escopo.

COCOMO II (Constructive Cost Model II) é um método de previsão de custos de software (Barry W. Boehm et al., 2000), focado no esforço em pessoas-mês necessários para terminar um projeto. É um método baseado em estatísticas recolhidas em centenas de projetos de software. Por ser um método totalmente aberto, uma empresa pode calibrar o método para refletir o seu ambiente de desenvolvimento.

Atualmente é possível conseguir gratuitamente um software COCOMO, o que facilita o cálculo de custos de projeto de sistemas de informação<sup>1</sup>.

Neste texto serão usadas as fórmulas e constantes do *COCOMO II Model Definition Manual version 1.4*(B. Boehm, Abts e Brad Clark, 1997).

<sup>1</sup>Uma lista de softwares na rede pode ser encontrada em <https://sites.google.com/cos.ufrj.br/livroanalisedesistemas>

## 16.1. Dois Modelos em Um

O método COCOMO II possui dois modelos, um chamado *Early Design Model*, adequado ao momento inicial de estimativa do projeto, e outro chamado *Post Architecture Model*, adequado ao momento onde a arquitetura do ciclo de vida de software está desenvolvida.

O *Early Design Model* usa como métrica de entrada a contagem de pontos de função não ajustada, e produz como métrica de saída o esforço necessário para completar o projeto.

## 16.2. Esforço em Função do Tamanho

O modelo básico do COCOMO II calcula o esforço em Pessoas-Mês em função do tamanho do software. As fórmulas, porém, se baseiam no tamanho do software em SLOCs, logo deve ser usada uma tabela de conversão de Pontos de Função para linhas de código, como a mostrada na Tabela 16.1.

$$PM_{\text{NOMINAL}} = A \times (\text{Size})^B \quad (16.1)$$

Onde  $A$  captura o esforço multiplicativo do aumento de tamanho e o expoente  $B$  captura as economias, e deseconomias, de escala que são encontradas em projetos de tamanhos diferentes(B. Boehm, Abts e Brad Clark, 1997). A Figura 16.1 exemplifica o efeito dessas constantes.

Linguagem	SLOC/PF
Assembly (com macros)	213
C	128
C++	128
Pascal	91
Planilha Eletrônica	6

Tabela 16.1.: Conversão de Pontos de Função não ajustados para linhas de código.

Fonte: (B. Boehm, Abts e Brad Clark, 1997)

### 16.3. Existe uma Relação Entre Esforço e Tempo

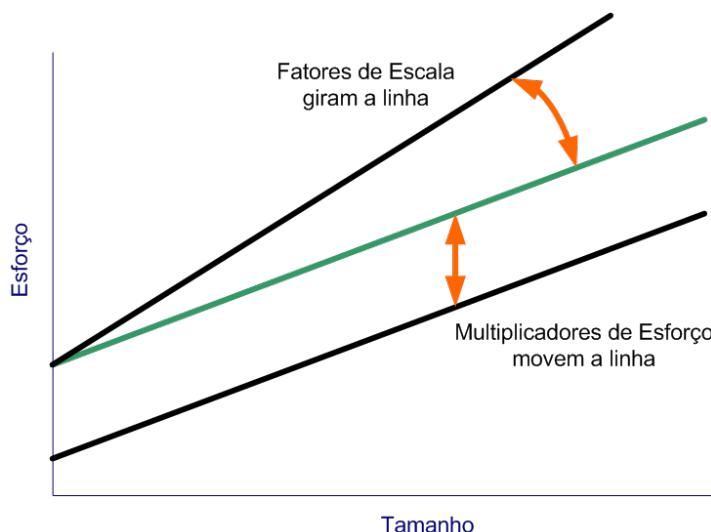


Figura 16.1.: Efeitos das constantes A e B na relação entre tamanho e esforço

## 16.3. Existe uma Relação Entre Esforço e Tempo

Um contribuição importante do método COCOMO II é determinar que existe uma relação entre o esforço necessário para desenvolver software e o tempo necessário para completar esse esforço. Antes mesmo de conhecer esse método, é importante deixar claro essa relação, pois muitos gerentes acreditam que podem espremer o tempo necessário para o desenvolvimento de um produto de software, o que leva as bastante conhecidas “Marchas da Morte” e, é claro, projetos atrasados, porque foram planejados de forma errada (Yourdon, 1994; Yourdon e Becker, 1997).

A suposição de que basta colocar mais gente, ou trabalhar mais horas, é suficiente para acelerar o tempo de desenvolvimento de um projeto, pode ser facilmente contestada com exemplos absurdos. Se esperamos que um software pode ser feito por 1 pessoa em 8 meses, é razoável imaginar, apesar de estar tecnicamente errado, que 2 pessoas poderiam fazê-lo em 4 meses. Porém, muito dificilmente 16 pessoas poderiam fazê-lo em 15 dias. Esse tipo de conta foi chamado de “Mito do Homem-Mês” (Brooks, 1995).

A relação entre o tempo de desenvolvimento e o esforço necessário, que apresentamos em sua forma completa a seguir, é parte importante do modelo COCOMO

$$T_{dev} = [C \times PM_{NS})^{D+0,2\times(E-B)}] \times \frac{SCED\%}{100} \quad (16.2)$$

Nessa fórmula  $PM_{NS}$  é a quantidade nominal de pessoas-mês , SCED% é a compressão necessária no tempo de desenvolvimento,  $B,C$  e  $D$  são constantes calibráveis e  $E$  é um coeficiente calculado a partir de fatores de escala.

Os valores das constantes estão na tabela

Tabela 16.2.: Valores das constantes no modelo COCOMO II padrão

A	2.94
B	0.91
C	3.67
D	0.28

O principal significado dessa fórmula pode ser associado ao fato que um projeto de desenvolvimento não pode ser comprimido, no tempo, além de certos limites. Coloquialmente, dizemos que software, como um bebê em gestação, pode até ficar bem feito se apressarmos o processo, porém não adianta pegar nove mães para fazer um bebê em um mês.

### 16.3.1. Cálculo dos Fatores de Escala

O valor do coeficiente  $E$  é calculado pela equação:

$$E = B + 0.01 \times \sum_{j=1}^N SF_j \quad (16.3)$$

Na fase inicial do projeto, os fatores de escala são 5 ( $N = 5$ ), e representam a precedência do sistema (PREC), a flexibilidade do projeto (FLEX), o risco do projeto e da arquitetura (RESL), a coesão da equipe (TEAM) e a maturidade do processo (PMAT). Cada um desses fatores de escala pode variar entre os conceitos:

- *Very Low* (muito baixa);
- *Low* (baixa);
- *Nominal* (nominal ou média);
- *High* (alta);
- *Very High* (muito alta), e
- *Extra High* (extremamente alta).

Originalmente, esses valores deveriam variar entre 5 (VL) e 0 (EH), porém foram corrigidos estatisticamente. Em situação nominal para todos os casos, o valor de E seria a 1,06, porém o software COCOMO usa outros valores, calibrados, o valor de E acaba sendo 1,10.

$$PM = A \times \prod_{j=1}^{17} c_i \times \left( \left( 1 + \frac{BRAK}{100} \right) \times PM_{NS} \right)^E \quad (16.4)$$

$$T_{DEV} = 3,67 \times (PM_{NS})^{0,32} \quad (16.5)$$

$$PM = 2,94 \times MLDC^{1,1} \quad (16.6)$$

## Bibliografia

- Adiel Teixeira de Almeida Danielle Morais, Hannu Nurmi (2019). *Systems, Procedures And Voting Rules In Context: A Primer For Voting Rule Selection*. Advances In Group Decision And Negotiation Vol. 9. Springer (ver p. 44).
- Adzic, Gojko e David Evans (2014). *Fifty Quick Ideas to Improve your User Stories*. Neuri Consulting (ver pp. 198, 200, 201).
- Albrecht, A.J (1979). “Measuring Application Development Productivity”. Em: *Proc. IBM Aplic. Dev. Symposium*. Monterey, CA, pp. 89–92 (ver p. 227).
- Almquist, Eric, Jamie Cleghorn e Lori Sherer (2018). “The B2B Elements of Value”. *Harvard Business Review*, pp. 72–81. URL: <https://hbr.org/2018/03/the-b2b-elements-of-value> (acesso em 09/02/2020) (ver pp. 37, 39, 40, 55).
- Almquist, Eric, John Senior e Nicolas Bloch (1 de set. de 2016). “The Elements of Value”. *Harvard Business Review*, pp. 46–53. URL: <https://hbr.org/2016/09/the-elements-of-value> (acesso em 09/02/2020) (ver pp. 36, 37, 55).
- Amy, Douglas J. (2000). *Behind the Ballot Box: A Citizen’s Guide to Voting Systems*. Greenwood Publishing Group (ver p. 44).
- Andersen, Erling S, Kristoffer V Grude e Tor Haug (2009). *Goal Directed Project Management: Effective techniques and strategies*. 4<sup>a</sup> ed. London: Kogan Page (ver p. 83).
- Anklesaria, Jimmy (2008). *Supply Chain Cost Management. The AIM & DRIVE Process for Achieving Extraordinary Results*. New York: Amacom - American Management Association (ver pp. 49, 50).
- Beck, Kent (29 de set. de 1999). *eXtreme Programming eXplained: Embrace Change*. 1<sup>a</sup> ed. USA: Addison-Wesley Longman Publishing Co., Inc. (ver p. 198).
- Beck, Kent, Alistair Cockburn et al. (10 de nov. de 2014). *User Story And Use Case Comparison*. URL: <http://wiki.c2.com/?UserStoryAndUseCaseComparison> (acesso em 13/01/2020) (ver p. 198).

## Bibliografia

- Beck, Kent, Martha Roden et al. (8 de jan. de 2014). *User Story*. URL: <http://wiki.c2.com/?UserStory> (acesso em 13/01/2020) (ver p. 198).
- Belchior, Arnaldo Dias (1997). “Um Modelo Fuzzy para Avaliação da Qualidade de Software”. D.Sc. Tese de dout. Rio de Janeiro, Brasil: Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ. 1-185 (ver pp. x, 91, 93, 95).
- Bellovin, Steve (11 de ago. de 1997). *Software error may have contributed to Guam crash*. Ed. por Peter G. Neumann. The RISKS Digest Forum on Risks to the Public in Computers, Related Systems ACM Committee on Computers e Public Policy. URL: %5Curl%7Bhttp://catless.ncl.ac.uk/Risks/19/29#subj1%7D (acesso em 20/12/2019) (ver p. 17).
- BeSeen (17 de ago. de 2015). *The value triangle – managing customers' expectations*. BeSeen. URL: <https://www.besehen-marketing.co.uk/marketing-blog/the-value-triangle> (acesso em 09/02/2020) (ver p. 54).
- Biffl, Stefan et al., ed. (2006). *Value-Based Software Engineering*. Berlin, Heidelberg: Springer (ver p. 53).
- Blanchard, Kenneth e Spencer Johnson (1983). *The One Minute Manager*. 10th anniversary. Berkley Trade (ver p. 106).
- Boehm, B. W. (mai. de 1988). “A spiral model of software development and enhancement”. *Computer* 21.5, pp. 61–72. ISSN: 1558-0814. DOI: 10.1109/2.59 (ver p. 77).
- Boehm, Barry, Chris Abts e and Sunita Devnani-Chulani Brad Clark (1997). *COCOMO II Model Definition Manual version 1.4* (ver pp. xiv, 239, 240).
- Boehm, Barry e Hasan Kitapci (2006). “The WinWin Approach: Using a Requirements Negotiation Tool for Rationale Capture and Use”. Em: *Rationale Management in Software Engineering*. Ed. por Raymond Dutoit Allen H.and McCall, Ivan Mistrík e Barbara Paech. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 173–190. ISBN: 978-3-540-30998-7. DOI: 10.1007/978-3-540-30998-7\_8. URL: [https://doi.org/10.1007/978-3-540-30998-7\\_8](https://doi.org/10.1007/978-3-540-30998-7_8) (ver p. 77).
- Boehm, Barry W. et al. (2000). *Software Cost Estimation with Cocomo II with Cdrom*. 1st. USA: Prentice Hall PTR. ISBN: 0130266922 (ver pp. 54, 218, 219, 221, 224, 239).
- Brasil (2018). *Roteiro de Métricas de Software do SISP: versão 2.3*. Roteiro. Versão 2.3. Ministério do Planejamento, Desenvolvimento e Gestão. Secretaria de Tecnologia da Informação e Comunicação - Setic. (ver pp. 12, 228).
- Brooks, Frederick P. (1978). *The Mythical Man-Month. Essays on Software Engineering*. 1<sup>a</sup> ed. USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201006502 (ver p. 54).
- (1995). *The Mythical Man-Month (Anniversary Ed.) Essays on Software Engineering*. 2<sup>a</sup> ed. USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201835959 (ver pp. 219, 241).
- Cairncross, A. (1951). *Introduction to Economics*. Butterworth (ver pp. 32, 33).
- Carr, Nicholas G. (22 de jan. de 2005). “Does Not Compute”. *The New York Times*. ISSN: 0362-4331. URL: <http://www.nytimes.com/2005/01/22/opinion/does-not-compute.html> (acesso em 04/03/2017) (ver p. 17).
- Christel, M. e K. Kand (set. de 1992). *Issues in Requirements Elicitation*. Technical Report CMU/SEI-92-TR-012. Software Engineering Institute / CMU, p. 80. URL:

- [https://resources.sei.cmu.edu/asset\\_files/TechnicalReport/1992\\_005\\_001\\_16478.pdf](https://resources.sei.cmu.edu/asset_files/TechnicalReport/1992_005_001_16478.pdf) (acesso em 24/02/2020) (ver pp. xi, xiv, 129, 131–133).
- CMMI Product Team (nov. de 2010). *CMMI for Development, Version 1.3 Improving processes for developing better products and services*. CMU/SEI-2010-TR-033. URL: %5Curl%7Bhttp://cmmiinstitute.com/system/files/models/CMMI\_for\_Development\_v1.3.pdf%7D (ver p. 96).
- Cockburn, Alistair (jan. de 2000). *Writing Effective Use Cases*. Addison-Wesley (ver pp. xiv, 170, 172, 173, 178, 180–183, 188).
- Cohn, Mike (2004). *User Stories Applied: For Agile Software Development*. USA: Addison Wesley Longman Publishing Co., Inc. ISBN: 0321205685 (ver pp. 45, 124, 198, 199, 201–205).
- (2005). *Agile Estimating and Planning*. Prentice Hall (ver pp. 45, 205, 222, 223, 225, 226).
- CompTIA (nov. de 2019). *CompTIA IT Industry Outlook 2020: Taking the Next Step*. Report. Computing Technology Industry Association (CompTIA). URL: <https://comptiacdn.azureedge.net/webcontent/docs/default-source/research-reports/comptia-it-industry-outlook-2020.pdf> (ver p. 13).
- Conklin, Jeff (18 de nov. de 2005). *Dialogue Mapping: Building Shared Understanding of Wicked Problems*. 1 edition. Chichester, England ; Hoboken, NJ: Wiley. 266 pp. ISBN: 978-0-470-01768-5 (ver pp. 17, 18).
- Craigen, Dan, Susan Gerhart e Ted Ralston (1993). “An International Survey of Industrial Applications of Formal Methods”. Em: *Z User Workshop, London 1992*. Ed. por J. P. Bowen e J. E. Nicholls. London: Springer London, pp. 1–5 (ver p. 123).
- Czarnacka-Chrobot, Beata (jul. de 2012). “What Is the Cost of One IFPUG Method Function Point? – Case Study”. Em: *The 11th International Conference on Software Engineering Research and Practice (SERP’12), The 2012 World Congress in Computer Science, Computer Engineering & Applied Computing (WORLDCOMP’12)*. The 11th International Conference on Software Engineering Research and Practice (SERP’12), The 2012 World Congress in Computer Science, Computer Engineering & Applied Computing (WORLDCOMP’12), Las Vegas, Nevada, USA: CSREA Press (ver p. 12).
- Dal Zot, Wili e Manuela Longoni de Castro (2015). *Matemática Financeira: fundamentos e aplicações*. Porto Alegre: Bookman, p. 151 (ver p. 48).
- Dalkey, Norman Crolee (out. de 1967). *Delphi*. Technical Report P-3704. Santa Monica, California: RAND Corporation. URL: <https://www.rand.org/pubs/papers/P3704.html> (acesso em 16/01/2020) (ver pp. 44, 225).
- Defeo, Joseph e J.M. Juran (9 de jun. de 2010). *Juran’s Quality Handbook: The Complete Guide to Performance Excellence 6/e*. 6 edition. New York: McGraw-Hill Education. 1136 pp. ISBN: 978-0-07-162973-7 (ver p. 88).
- Dijkstra, Edsger W. (out. de 1972). “The Humble Programmer”. *Commun. ACM* 15.10, pp. 859–866. ISSN: 0001-0782. DOI: 10.1145/355604.361591. URL: <http://doi.acm.org/10.1145/355604.361591> (ver p. 17).
- Dimitrov, Dimitri (2020). *Software Project Estimation. Intelligent Forecasting, Project Control, and Cliente Relationship Management*. Apress (ver p. 223).

## Bibliografia

- Drucker, Peter F. (1974). *Management: Tasks, Responsibilities, Practices*. New York: Truman Talley Books (ver p. 36).
- Erdogmus, Hakan, John Favaro e Michael Halling (2006). “Valuation of Software Initiatives Under Uncertainty: Concepts, Issues, and Techniques”. Em: *Value-Based Software Engineering*. Ed. por Stefan Biffl et al. Berlin, Heidelberg: Springer (ver p. 53).
- Extra (18 de jun. de 2019). *Se quer levar mais de 10 quilos, pague, sem problema nenhum’, diz Bolsonaro sobre fim do despacho gratuito*. URL: <https://extra.globo.com/noticias/economia/se-quer-levar-mais-de-10-quilos-pague-sem-problema-nenhum-diz-bolsonaro-sobre-fim-do-despacho-gratuito-23747656.html> (acesso em 30/01/2020) (ver p. 61).
- Facetation (12 de mai. de 2015). *What is the history of the RACI chart?* URL: <http://facetation.blogspot.com/2015/05/what-is-history-of-raci-chart.html> (acesso em 06/10/2020) (ver p. 83).
- FATTO (2020). *Quanto pagar por um ponto de função?* URL: <https://docplayer.com.br/13499465-Quanto-pagar-por-um-ponto-de-funcao.html> (acesso em 02/02/2020) (ver p. 12).
- Fedotov, Alex (22 de fev. de 2019). *Septem Circumstantiae, five W's and H or 'six serving-men'*. URL: <https://alxfed.github.io/blog/posts/2019/02/22/Septem-Circumstantiae.html> (acesso em 08/01/2020) (ver p. 60).
- Flores, Fernando (2012). *Conversation for Action and Collected Essays: Instilling a Culture of Commitment in Working Relationships*. Ed. por Maria Flores Letelier. North Charleston, SC: CreateSpace Independent Publishing Platform (ver p. 62).
- Folha Online (29 de mai. de 2006). *Volkswagen anuncia recall de 123 mil veículos Gol, Fox e Kombi*. Folha Online - Dinheiro. URL: <http://www1.folha.uol.com.br/folha/dinheiro/ult91u108087.shtml> (acesso em 03/03/2017) (ver p. 16).
- Franceschini, Fiorenzo (2016). *Advanced Quality Function Deployment*. CRC Press, p. 208. ISBN: 9781420025439 (ver p. 40).
- Gane, Chris P. e Trish Sarson (1979). *Structured Systems Analysis: Tools and Techniques*. 1<sup>a</sup> ed. Prentice Hall Professional Technical Reference. ISBN: 0138545472 (ver pp. 53, 123, 124, 171).
- Garvin, David A. (1984). “What Does Product Quality Really Mean?” *MIT Sloan Management Review*. URL: <http://sloanreview.mit.edu/article/what-does-product-quality-really-mean/> (acesso em 06/03/2017) (ver pp. 90, 91).
- Gibbs, Wayt W. (set. de 1994). “Software’s Chronic Crisis”. *Scientific American*, pp. 86–100 (ver pp. 16, 18).
- Gillenson, Mark L. e Robert Goldberg (1984). *Strategic Planning, Systems Analysis, and Database Design: The Continuous Flow Approach*. John Wiley & Sons (ver p. 116).
- Gladys S. W. Lam, Ronald G. Ross and (out. de 2015). *Building Business Solutions: Business Analysis with Business Rules*. 2<sup>a</sup> ed. sponsored by IIBA: International Institute of Business Analysis. Business Rule Solutions LLC., p. 304 (ver p. 72).
- Gray, Dave, Sunni Brown e James Macanufo (2010). *Gamestorming*. Sebastopol, CA: O'Reilly (ver p. 45).
- Greenlaw, Steven A., David Shapiro e Timothy Taylor (2017). *Principles of Microeconomics for AP® Courses*. 2<sup>a</sup> ed. OpenStax, Rice University (ver p. 34).

- Group, Gartner (21 de mar. de 2014). *Gartner Says Worldwide Software Market Grew 4.8 Percent in 2013*. URL: %5Curl%7Bhttp://www.gartner.com/newsroom/id/2696317%7D (ver p. 13).
- Hastie, Shane e Stéphane Wojewoda (4 de out. de 2015). *Standish Group 2015 Chaos Report - Q&A with Jennifer Lynch*. URL: <https://www.infoq.com/articles/standish-chaos-2015/> (acesso em 15/12/2019) (ver p. 23).
- Hayes, Adam (25 de jun. de 2019). *Internal Rate of Return – IRR*. Investopedia. URL: <https://www.investopedia.com/terms/i/IRR.asp> (acesso em 08/02/2020) (ver pp. 49, 52).
- Head, Robert V. (out. de 2002). “Getting Sabre off the Ground”. *IEEE Annals of the History of Computing* 24 (4), pp. 32–39. DOI: 10.1109/MAHC.2002.1114868 (ver p. 11).
- Helmer-Hirschberg, Olaf (mar. de 1967). *Analysis of the Future: The Delphi Method*. Technical Report P-3558. Santa Monica, California: RAND Corporation. URL: <https://www.rand.org/pubs/papers/P3558.html> (acesso em 16/01/2020) (ver pp. 45, 225).
- Higgins, J.M. (1994). *101 Creative Problem Solving Techniques: The Handbook of New Ideas for Business*. New Management Publishing Company. ISBN: 9781883629007. URL: [https://books.google.com.br/books?id=Q1%5C\\_9LcYV03kC](https://books.google.com.br/books?id=Q1%5C_9LcYV03kC) (ver p. 64).
- Holanda Ferreira, Aurélio Buarque de (1986). *Novo Dicionário da Língua Portuguesa*. 2<sup>a</sup> ed. Nova Fronteira (ver p. 32).
- IEEE (25 de jun. de 1998). *Recommended Practice for Software Requirements Specifications*. Standard ISO/IEC/IEEE 830:1998. IEEE (ver pp. 128, 143, 144).
- (dez. de 2010). *ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary*. Standard 24765:2010(E). IEEE, pp. 1–418. DOI: 10.1109/IEEESTD.2010.5733835 (ver p. 122).
- (2017a). *IEEE Recommended Practice on Software Reliability*. Standard IEEE Std 1633<sup>TM</sup>-2016. New York: IEEE (ver p. 96).
- (2017b). *ISO/IEC/IEEE International Standard - Systems and software engineering– Vocabulary*. Standard ISO/IEC/IEEE 24765:2017(E). ISO/IEC/IEEE, pp. 1–541. DOI: 10.1109/IEEESTD.2017.8016712 (ver p. 10).
- (nov. de 2018). *ISO/IEC/IEEE International Standard - Systems and software engineering – Life cycle processes – Requirements engineering*. Standard 29148:2018. IEEE, pp. 1–104. DOI: 10.1109/IEEESTD.2018.85559686 (ver pp. 135, 137, 142–144).
- IEEE Computer Society (17 de jan. de 2014a). *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. Ed. por Pierre Bourque e Richard E. Fairley. 3 edition. IEEE Computer Society Press. 346 pp. ISBN: 978-0-7695-5166-1 (ver pp. ix, x, 21, 24, 25, 87–89, 97, 218).
- (2014b). *Software Engineering Competency Model Version 1.0 SWECOM*. New York: IEEE Computer Society (ver pp. 29, 30).
- IFPUG (2010). *Function Point Counting Practices Manual Release 4.3.1*. Standard. International Function Point Users Group (ver pp. 12, 45, 236).
- (2012). *The IFPUG Guide to IT and Software Measurement*. Auerbach Publications (ver p. 228).

## Bibliografia

- IIBA (2011). *Um guia para o Corpo de Conhecimento de Análise de Negócios (Guia BABOK) Version 2.0*. Ontário, Canadá: International Institute of Business Analysis (ver pp. 41, 55, 140).
- ISBS, International Software Benchmarking Standards Group - (2010). *Practical Software Project Estimation: A Toolkit for Estimating Software Development Effort & Duration*. Ed. por Peter Hill. McGraw Hill (ver pp. 223–226).
- ISO (15 de jun. de 2001). *Software Engineering – Product Quality – Part 1: Quality Model*. Standard ISO/IEC 9126-1:2001. Geneva, Switzerland: International Standards Organization (ver p. 88).
- (nov. de 2004). *Information technology — Process assessment — Part 1: Concepts and vocabulary*. Standard ISO/IEC 15504-1:2004. Geneva, Switzerland: ISO International Standards Organization (ver p. 96).
  - (set. de 2011). *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. Standard ISO/IEC 25010:2011(en). Geneva, Switzerland: ISO International Standards Organization (ver pp. x, 88, 91, 93, 94).
  - (1 de mar. de 2015a). *Information technology – Process assessment – Concepts and terminology*. Standard ISO/IEC 33001. Geneva, Switzerland: ISO International Standards Organization (ver p. 96).
  - (15 de set. de 2015b). *Quality management systems — Fundamentals and vocabulary*. Standard ISO 9000:2015. Geneva, Switzerland: ISO International Standards Organization, p. 51 (ver pp. 88, 99).
  - (2017). *Systems And Software Engineering. Software Life Cycle Processes (British Standard)*. Standard BS ISO/IEC/IEEE 12207:2017. International Standards Organization (ver pp. 96, 99, 213).
- ISO/IEC/IEEE (2017). “ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary”. *ISO/IEC/IEEE 24765:2017(E)*, pp. 1–541. DOI: 10.1109/IEEESTD.2017.8016712 (ver pp. 211, 212).
- Jacobson, Ivar (dez. de 1987). “Object-Oriented Development in an Industrial Environment”. *SIGPLAN Not.* 22.12, pp. 183–191. ISSN: 0362-1340. DOI: 10.1145/38807.38824. URL: <https://doi.org/10.1145/38807.38824> (ver p. 170).
- (1 de ago. de 2004). “Use cases – Yesterday, today, and tomorrow”. *Software & Systems Modeling* 3.3, pp. 210–220. ISSN: 1619-1374. DOI: 10.1007/s10270-004-0060-3. URL: <https://doi.org/10.1007/s10270-004-0060-3> (ver p. 171).
- Jacobson, Ivar, Magnus Christerson et al. (1992). *Object-oriented software engineering: a use case driven approach*. ACM Press Series. Reading: Addison-Wesley. ISBN: 9780201544350 (ver pp. 124, 170).
- Jacobson, Ivar, Harold ”Bud” Lawson e Pan-Wei Ng (19 de jul. de 2019). *The Essentials of Modern Software Engineering: Free the Practices from the Method Prisons!* ACM Books. 400 pp. ISBN: 978-1-947487-24-6 (ver pp. 26, 81, 83, 104, 105, 198, 204–206).
- Jacobson, Ivar, Paul E. McMahon e Roland Racko (2015). *24 Questions SEMAT and Essence: The Why's, What's and How's to See the Difference*. Rel. técn. Ivar Jacobson International (ver p. 26).

- Jacobson, Ivar, Pan-Wei Ng et al. (26 de jan. de 2013). *The Essence of Software Engineering: Applying the SEMAT Kernel*. 1 edition. Upper Saddle River, NJ: Addison-Wesley Professional. 352 pp. ISBN: 978-0-321-88595-1 (ver pp. ix, x, 26–28, 70, 76).
- Jacobson, Ivar, Ian Spence e Kurt Bittner (dez. de 2011). *Use-Case 2.0. The Guide to Succeeding with Use Cases*. White Paper. Ivar Jacobson International S.A. (ver pp. 124, 172).
- Jeffries, Ron (30 de ago. de 2001). *Essential XP: Card, Conversation, Confirmation*. URL: <https://ronjeffries.com/xprog/articles/expcardconversationconfirmation/> (acesso em 13/01/2020) (ver p. 199).
- Johnson, Phil (8 de ago. de 2012). *Curiosity about lines of code*. ITworld. URL: <http://www.itworld.com/article/2725085/big-data/curiosity-about-lines-of-code.html> (acesso em 04/03/2017) (ver p. 18).
- Jones, C. e O. Bonsignour (2012). *The Economics of Software Quality*. Addison-Wesley. ISBN: 9780132582209 (ver pp. xiii, 97, 98).
- Jones, Caper (abr. de 2005). “Software Cost Estimating Methods for Large Projects”. *Crosstalk The Journal of Defense Software Engineering* (April 2005). Disponível via WayBack Machine. URL: <http://www.stsc.hill.af.mil/crosstalk/2005/04/0504Jones.html> (ver p. 140).
- Kelley Jr, J. E. e M. R. Walker (1961). “Critical-path planning and scheduling”. Em: *Proceedings of the Eastern Joint Computer Conference*, pp. 160–173 (ver p. 161).
- Kendall, Kenneth E. e Julie E. Kendall (2013). *Systems Analysis and Design*. 9<sup>a</sup> ed. Pearson (ver p. 118).
- Kenton, Will (25 de jun. de 2019). *Net Present Value (NPV)*. Investopedia. URL: <https://www.investopedia.com/terms/n/npv.asp> (acesso em 08/02/2020) (ver p. 52).
- Kerzner, Harold (2017). *Project Management. A system approach to planning, scheduling and controlling*. 12<sup>a</sup> ed. Hoboken, New Jersey: John Wiley & Sons (ver pp. 54, 152–154, 161, 162).
- Kim, W.C. e R. Mauborgne (2005). *A Estratégia Do Oceano Azul*. Elsevier. ISBN: 9788535215243 (ver p. 40).
- King, J. E. e Michael McLure (2014). *History of the Concept of Value*. Discussion Paper 14.06. The University of Western Australia, Department of Economics. URL: <https://ideas.repec.org/p/uwa/wpaper/14-06.html> (ver p. 33).
- Kotler, Philip, Gary Armstrong et al. (2017). *Principles of Marketing*. 7th European Edition. Pearsppm (ver p. 35).
- Kotler, Philip e Kevin Lana Keller (2012). *Marketing Management*. 14<sup>a</sup> ed. Boston: Prentice Hall (ver p. 35).
- (2013). *Administração de Marketing*. 14<sup>a</sup> ed. São Paulo: Pearson (ver p. 36).
- Krasner, Herb (1 de jan. de 2021). *The Cost of Poor Software Quality in the US: A 2020 Report*. type. CISQ Consortium for Information & Software Quality. 46 pp. URL: <https://www.it-cisq.org/pdf/CPSQ-2020-report.pdf> (acesso em 26/06/2021) (ver p. 16).
- Krugman, Paul e Robin Wells (2013). *Microeconomics*. 3<sup>a</sup> ed. New York: Worth Publishers (ver pp. 33, 34).

## Bibliografia

- Ladkin, Peter (7 de mai. de 1996). *The Cali and Puerto Plata B757 Crashes*. Ed. por Peter G. Neumann. Volume 18 Issue 10. The RISKS Digest Forum on Risks to the Public in Computers, Related Systems ACM Committee on Computers e Public Policy. URL: %5Curl%7Bhttp://catless.ncl.ac.uk/Risks/18/10#subj%7D (acesso em 20/12/2019) (ver p. 17).
- Leffingwell, Dean e Don Widrig (1999). *Managing Software Requirements: A Unified Approach*. USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201615932 (ver p. 41).
- Levenson, Nancy G. e Clark S. Turner (jul. de 1993). “An Investigation of the Therac-25 Accidents”. *Computer* 26.7, pp. 18–41 (ver p. 16).
- Levere, Jane L. (16 de mar. de 2001). “TECHNOLOGY; Electronic Data Systems to Buy Sabre Airline Computer Unit”. *The New York Times*, Section C, Page 8. URL: <https://www.nytimes.com/2001/03/16/business/technology-electronic-data-systems-to-buy-sabre-airline-computer-unit.html> (acesso em 02/02/2020) (ver p. 11).
- Lions, J. L. (19 de jul. de 1996). *ARIANE 5 Flight 501 Failure Report by the Inquiry Board*. Report. Paris: European Space Agency (ver p. 16).
- Malcolm, D. G. et al. (1959). “Application of a technique for research and development program evaluation”. *Operations research* 7.5, pp. 646–669 (ver p. 161).
- Maximini, Dominik (2018). *The Scrum Culture. Introducing Agile Methos in Organizations*. 2<sup>a</sup> ed. Management for Professionals. Springer (ver p. 226).
- McCall, J.A., P. K. Richards e G.F. Walters (nov. de 1977). *Factors in Software Quality: Concepts and Definitions of Sofware Quality*. Rel. técn. RADC-TR-77-369. General Electric. URL: %5Curl%7Bhttp://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA049055%7D (ver pp. x, 91, 92).
- McMenamin, Stephen M. e John F. Palmer (1984). *Essential Systems Analysis*. USA: Yourdon Press. ISBN: 0917072308 (ver pp. 104, 105, 123, 134, 135).
- Meskanen, T e H Nurmi (2006). “Distance from Consensus: A Theme and Variations”. Em: *Mathematics and Democracy. Studies in Choice and Welfare*. Ed. por Pukelsheim F. Simeone B. Berlin, Heidelberg: Springer. DOI: 10.1007/3-540-35605-3\_9 (ver p. 45).
- Mitchell, Ronald K., Bradley R. Agle e Donna J. Wood (1997). “Toward a Theory of Stakeholder Identification and Salience: Defining the Principle of Who and What Really Counts”. *The Academy of Management Review* 22.4, pp. 853–886. ISSN: 0363-7425. DOI: 10.2307/259247. URL: %5Curl%7Bhttp://www.jstor.org/stable/259247%7D (acesso em 15/01/2017) (ver pp. x, 81, 82).
- Moorman, Jan (9 de out. de 2012). *Leveraging the Kano Model for Optimal Results*. Article No :882. UX Magazine. URL: <https://uxmag.com/articles/leveraging-the-kano-model-for-optimal-results> (acesso em 08/02/2020) (ver pp. xiii, 42, 43, 55).
- Naur, Peter e Brian Randell, ed. (jan. de 1969). *Software Engineering. Report on a conference sponsored by theNATO SCIENCE COMMITTEE*Garmisch, Germany, 7th to 11th October 1968 (ver p. 17).
- Niquette, Paul (2006). *Softword: Provenance for the Word 'Software'*. URL: <http://niquette.com/books/softword/tocsoft.html> (acesso em 02/02/2017) (ver p. 10).

- North, Dan (16 de nov. de 2013). *User Story Template*. URL: <http://wiki.c2.com/?UserStoryTemplate> (acesso em 13/01/2020) (ver p. 200).
- OMG (5 de dez. de 2017). *OMG®Unified Modeling Language®(OMG UML®)*. version 2.5.1. specification formal/2017-12-05. Object Management Group (ver pp. 124, 171, 173, 180, 189, 194).
- (out. de 2018). *Essence – Kernel and Language for Software Engineering Methods: Version 1.2*. specification. Object Management Group. URL: %5Curl%7Bhttps://www.omg.org/spec/Essence/%7D (ver p. 104).
- Park, Robert E. (set. de 1992). *Software Size Measurement: A Framework for Counting Source Statements*. Technical Report CMU/SEI-92-TR-020 ESC-TR-92-020. Software Engineering Institute, Carnegie Mellon University. URL: [https://resources.sei.cmu.edu/asset\\_files/TechnicalReport/1992\\_005\\_001\\_16082.pdf](https://resources.sei.cmu.edu/asset_files/TechnicalReport/1992_005_001_16082.pdf) (acesso em 16/01/2020) (ver p. 222).
- Patton, Jeff e Peter Economy (2014a). *User Story Mapping: Discover the Whole Story, Build the Right Product*. 1st. O'Reilly Media, Inc. ISBN: 1491904909 (ver pp. xii, 200).
- (2014b). *User Story Mapping: Discover the Whole Story, Build the Right Product*. 1st. O'Reilly Media, Inc. (ver p. 159).
- PMI (2017). *Um Guia do Conhecimento em Gerenciamento de Projetos (Guia PM-BOK®)*. Sexta Edição. Newtown Square, PA: Project Management Institute. ISBN: 9788502223745 (ver pp. x, 64, 71–73, 75, 76, 80, 81, 83, 151–154, 157–159, 218, 225).
- Pressman, Roger S. e Bruce Maxim (23 de jan. de 2014). *Software Engineering: A Practitioner's Approach*. 8 edition. New York, NY: McGraw-Hill Education. 976 pp. ISBN: 978-0-07-802212-8 (ver pp. x, 10, 20–22, 87, 89, 218).
- (2019). *Software Engineering: A Practitioner's Approach*. 9<sup>a</sup> ed. New York, NY: McGraw-Hill. 976 pp. ISBN: 978-1-260-54800-6 (ver p. 211).
- Ricardo, David (1996). *Princípios de Economia Política e Tributação*. Os Economistas. texto original de 1821. São Paulo: Editora Nova Cultural Ltda. (ver p. 33).
- Robertson, James e Suzanne Robertson (1998). *Complete Systems Analysis*. New York: Dorser House (ver p. 41).
- (2006). *Mastering the Requirements Process*. 2<sup>a</sup> ed. Addison-Wesley Professional. ISBN: 0321419499 (ver pp. xi, 43, 55, 124, 128, 129, 137).
- Rösler, Wolfram (1994). *The Hello World Collection*. URL: <http://helloworldcollection.de/> (acesso em 15/01/2020) (ver pp. 220, 221).
- Ross, Timothy J. (2017). *Fuzzy logic with engineering applications*. Southern Gate, Chichester, West Sussex, United Kingdom: John Wiley & Sons (ver p. 45).
- Rubin, Kenneth S. (2013). *Essential Scrum: A Practical Guide to the Most Popular Agile Process*. Pearson Education (ver pp. 83, 223).
- Ruble, David A. (1997). *Practical Analysis & Design for Client/Server and GUI Systems*. Upper Saddle River: Yourdon Press (ver pp. 53, 104, 171).
- Satpathy, Tridibesh et al. (2016). *A Guide to the SCRUM BODY OF KNOWLEDGE (SBOK™ Guide)*. A Comprehensive Guide to Deliver Projects using Scrum. 2016<sup>a</sup> ed. SCRUMStudy, VMEdu, Inc (ver pp. 41, 49, 54, 153).
- Savage, Neil (jun. de 2020). “An Animating Spirit”. *Communications of the ACM* 63.6, pp. 10–12. DOI: 10.1145/3392520 (ver p. 14).

## Bibliografia

- SEMAT Inc. (2019). *ESSENCE User Guide*. URL: %5Curl%7Bhttp://semat.org/essence-user-guide%7D (acesso em 20/12/2019) (ver p. 26).
- Serrat, Olivier (2017). “The Five Whys Technique”. Em: *Knowledge Solutions: Tools, Methods, and Approaches to Drive Organizational Performance*. Singapore: Springer Singapore, pp. 307–310. ISBN: 978-981-10-0983-9. DOI: 10.1007/978-981-10-0983-9\_32. URL: https://doi.org/10.1007/978-981-10-0983-9\_32 (acesso em 10/02/2020) (ver p. 63).
- Sloan, Michael C. (2010). “Aristotle’s Nicomachean Ethics as the Original Locus for the Septem Circumstantiae”. *Classical Philology* 105.3, pp. 236–251. ISSN: 0009837X, 1546072X. URL: http://www.jstor.org/stable/10.1086/656196 (ver p. 60).
- Smith, Adam (1996). *A Riqueza das Nações. Investigaçāo sobre sua natureza e suas causas*. Os Economistas. texto original de 1776. São Paulo: Editora Nova Cultural Ltda. (ver p. 33).
- (2003). *The Wealth of Nations*. texto original de 1854. Bantam Classics. ISBN: 0553585975 (ver p. 33).
- Sommerville, Ian (3 de abr. de 2015). *Software Engineering*. 10 edition. Boston: Pearson. 816 pp. ISBN: 978-0-13-394303-0 (ver pp. xi, 124, 127, 130).
- Sommerville, Ian e Pete Sawyer (1997). *Requirements Engineering: A Good Practice Guide*. 1st. USA: John Wiley & Sons, Inc. ISBN: 0471974447 (ver p. 123).
- Sowa, J.F. e J.A. Zachman (1992). “Extending and Formalizing the Framework for Information Systems Arquitecture”. *IBM Systems Journal* 31.3, p. 590 (ver p. 62).
- Sr., Lance B. Coleman (2015). *The Customer-Driven Organization: Employing the Kano Model*. 1<sup>a</sup> ed. Productivity Press (ver p. 55).
- Stapenhurst, Tim (2009). *The Benchmarking Book*. Butterworth. ISBN: 0750677775 (ver p. 107).
- Stapleton, Jennifer, ed. (2003). *DSDM: Business Focused Development*. 2<sup>a</sup> ed. London: DSDM Consortium, Addison Wesley (ver p. 42).
- Strathern, Paul (2003). *Uma Breve História da Economia*. Zahar (ver p. 33).
- Telles, M. e Y. Hsieh (2001). *The Science of Debugging*. Sebastopol, CA: O'Reilly Media (ver p. 212).
- The Standish Group (1994). *The Chaos Report 1994*. Report. The Standish Group (ver p. 21).
- (2015). *Chaos Report 2015*. Report. Standish Group (ver pp. xiii, 19, 20, 23, 24).
- Tricentis (2019). *Software FAILS Watch*. White Paper. Versão 5th Edition. Tricentis (ver p. 16).
- Tukey, John W. (1958). “The Teaching of Concrete Mathematics”. *The American Mathematical Monthly* 65.1, pp. 1–9. ISSN: 0002-9890. DOI: 10.2307/2310294. URL: www.jstor.org/stable/2310294 (acesso em 20/12/2019) (ver p. 10).
- Vaughan-Nichols, Steven J. (7 de nov. de 2017). *MINIX: Intel’s hidden in-chip operating system*. URL: https://www.zdnet.com/article/minix-intels-hidden-in-chip-operating-system/ (acesso em 02/02/2020) (ver p. 10).
- Venners, B. (8 de dez. de 2003). *Design by Contract: A Conversation with Bertrand Meyer*. URL: www.artima.com/intv/contracts.html (ver p. 97).

- Volare (2020). *Prioritisation Analysis*. URL: <https://www.volere.org/prioritisation-analysis/> (acesso em 24/02/2020) (ver p. 140).
- Wake, Bill (17 de ago. de 2003). *INVEST in Good Stories, and SMART Tasks*. URL: <https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/> (acesso em 14/01/2020) (ver pp. 202, 204).
- (8 de fev. de 2012). *Independent Stories in the INVEST Model*. URL: <https://xp123.com/articles/independent-stories-in-the-invest-model/> (acesso em 14/01/2020) (ver pp. 203, 204).
- Wiegers, Karl (mai. de 1999). “Writing Quality Requirements”. *Software Development* 7.5, pp. 44–48. ISSN: 1070-8588 (ver p. 146).
- Wiegers, Karl E. e Joy Beatty (2013). *Software Requirements*. 3<sup>a</sup> ed. Redmond, WA, USA: Microsoft Press (ver pp. xiv, 80, 124, 129, 130, 134, 199).
- Yourdon, Edward (1994). *Decline and Fall of the American Programmer*. 1st. USA: Prentice Hall PTR. ISBN: 013191958X (ver p. 241).
- Yourdon, Edward e Paul D. Becker (1997). *Death March: The Complete Software Developer's Guide to Surviving Mission Impossible Projects*. USA: Prentice Hall PTR. ISBN: 0137483104 (ver p. 241).
- Zachman, J.A. (1987). “A Framework for Information Systems Architecture”. *IBM Systems Journal* 26.3, p. 276 (ver p. 62).



# Índice Remissivo

*RFP*, **123**

*Request for Proposal*, **123**

*Work Breakdown Structure*, **158**

*future value*, **48**

*present value*, **48**

100-pontos, **41**

5W2H, **59**

5w2h, **59**

alpha, **26**

arquivos de interface externos, **230**

arquivos lógicos internos, **230**

ator, **173**

ator principal, **172, 173**

bem sucedidos, **19**

benefício, **53**

bug, **212**

Capability Maturity Model, **96**

Capability Maturity Model Integration,  
**96**

caso de uso, **172**

diagrama, **188**

caso de uso 2.0, **172**

casos de uso, **169**

causa raiz, **105**

cenário, **173, 175**

alternativo, **173**

principal, **173**

cenário alternativo, **177**

cenário principal, **175**

CMM, **96**

CMMI, **96**

com problemas, **19**

condição, **173**

consultas externas, **230**

Controlgramas de Gantt, **159**

CPM, **160**

crash, **212**

custo de oportunidade, **34**

defeito (defect), **212**

demandas, **35**

estados, **35**

desejo, **35**

diagrama de caso de uso, **188**

Diagrama de Causa Raiz, **111**

Diagrama de Espinha de Peixe, **111**

Diagramas de Rede, **161**

dinheiro de brinquedo, **41**

EAP, **158**

elicitação, **129**

Engenharia de Software, 9, **21**

entradas externas, **230**

## ÍNDICE REMISSIVO

- entrega intermediária, 152  
erro (error), **212**  
erro (mistake), **212**  
esforço, **219**  
especificação de requisitos, **122**  
estimativa de tamanho software, 217  
Estrutura Analítica de Projeto, **158**  
estruturas de dados, **10**
- falha (failure), **212**  
falta ou defeito (fault), **212**  
fracassos, **19**  
Framework Cynefin, **4**  
Framework de Zachman, 62  
funcionalidade, **19**  
função utilidade, **33**
- garantias, **184**
- hardware, **10**  
história do usuário, 197, **198**  
histórias complexas, **205**  
histórias compostas, **205**
- IMC, **164**  
IMT, **165**  
informação, **10**  
instrução, 10  
interesse, **77**, 77  
Início Mais Cedo, **164**  
Início Mais Tarde, **165**  
IRACIS, **53**  
ISO 25000, **88**  
ISO 25010, **91**  
ISO 9126, **88**
- juro, **48**
- Kano, 42  
kernel, 26
- legitimidade, **81**  
lista de atores, **188**
- markup, **33**  
Matriz de Engajamento, **79**
- menor produto viável, **204**  
modo de trabalho, 26  
moeda virtual, **140**  
montante, **48**  
MoSCoW, **41**, **140**  
MTBF, **127**  
Mundo Real, 3  
método, **26**
- narrativa, 176  
numerada, 176  
necessidade, 35
- objetivo, **77**, 77, 173  
oportunidade, 26, 103, **104**  
    tecnológica, 117
- oportunidades  
    de negócio, 117
- orçamento, **19**
- pacote de trabalho, 159  
pagamento, **48**  
parte interessada, 183  
partes interessadas, **26**, **70**, **153**  
PERT, **160**  
PERT/CPM, 160  
planning poker, 223, 225  
poder, **81**  
pontos de função, 227  
Pontos de história, **222**  
pontos de história, **225**  
prazo, **19**, **48**  
prestação, **48**  
principal, **48**  
princípio da diminuição da utilidade  
    marginal, **33**  
Princípio de Pareto, **107**  
problema, 103  
problemas de negócio, **106**  
problemas funcionais, **106**  
problemas operacionais, **106**  
projeto, **151**  
prática, 26  
pré-condição, 184  
pós-condição, 184

## ÍNDICE REMISSIVO

- qualidade, **88**  
qualidade de software, 87  
**QWAN, 90**  
redes de precedência, 161  
remuneração, **48**  
requisito, 26, **122**  
    arbitrário, 134, 135  
    do negócio, 134  
    do sistema, 134  
    do software, 134  
    do usuário, 134  
    elicitação, 129  
    especificação, 122  
    falso, 134  
    tecnológico, 134, 135  
requisito de software, **198**  
requisitos de desempenho, 128  
requisitos de interface, 128  
requisitos funcionais, **126**, 126  
requisitos não funcionais, **126**, 126  
restrições, **126**  
  
saldo, **48**  
saliência, **81**  
satisfação, **36**  
saídas externas, **230**  
sistema de software, **26**  
sobreposição, **203**  
software, **10**, 10  
stakeholders, **70**  
  
tamanho do software, 217, 219  
taxa, **48**  
taxa de juros, **48**  
Testes, 211  
testes regressivos, **213**  
time, **26**  
**TMC, 164**  
**TMT, 165**  
trabalho, **26**  
**Término Mais Cedo, 164**  
**Término Mais Tarde, 165**  
  
urgência, **81**  
user story, 197  
usuário, **73**  
usuário externo, **74**  
usuário final, **73**  
usuário interno, **74**  
usuário onisciente, **73**  
utilidade, 33  
  
valor, **11**, 31, **32**, 32, 36, **53**, 53  
valor atual, **48**  
valor de resgate, **48**  
valor descontado, **48**  
valor futuro, **48**  
valor presente, **48**  
**Volere, 140**  
  
**WBS, 158**  
wicked problems, **17**