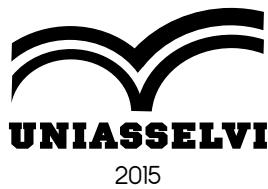


ENGENHARIA E PROJETO DE SOFTWARE

Prof. Pedro Sidnei Zanchett





Copyright © UNIASSELVI 2015

Elaboração:
Prof. Pedro Sidnei Zanchett

Revisão, Diagramação e Produção:
Centro Universitário Leonardo da Vinci – UNIASSELVI

Ficha catalográfica elaborada na fonte pela Biblioteca Dante Alighieri
UNIASSELVI – Indaial.

005.102
Z27e Zanchett, Pedro Sidnei

Engenharia e projeto de software/ Pedro Sidnei Zanchett.
Indaial : UNIASSELVI, 2015.

267 p. : il.

ISBN 978-85-7830-921-3

1. Engenharia de software.
- I. Centro Universitário Leonardo Da Vinci.

APRESENTAÇÃO

Prezado(a) acadêmico(a)! Seja bem-vindo(a) à disciplina de Engenharia e Projeto de *Software*.

Este Caderno de Estudos foi elaborado com o intuito de contribuir e aprimorar o seu conhecimento acerca destas três unidades principais: Engenharia de *Software*, Gestão de Projeto de *Software* e Qualidade de *Software*. Importantes áreas de conhecimento da computação/informática voltada para a especificação, desenvolvimento e manutenção de sistemas de *software*.

Um desafio constante é como desenvolver *softwares* cada vez maiores e integrados com outros *softwares* e, ainda, ter qualidade e produtividade adequada para atender às necessidades dos clientes e cumprir os prazos e custos dos projetos. O foco desta disciplina está em fornecer a você uma visão sistêmica da Engenharia e Projeto de *Software*, onde estudará importantes conceitos relacionados a aspectos técnicos de construção de *software*, bem como os aspectos gerenciais que envolvem seu processo de desenvolvimento a fim de aperfeiçoar sua visão e estar mais preparado para lidar com os problemas do dia a dia.

Não é fácil desenvolver *softwares* de qualidade, por isso, esta disciplina se torna tão necessária. Os conceitos aqui apresentados representam o amadurecimento das técnicas, métodos, ferramentas e atividades utilizadas ao longo dos anos no ambiente de desenvolvimento do *software*, onde o planejamento, o controle e a produtividade poderão ser obtidos com maior qualidade e sucesso.

Aproveitamos esse momento para destacar que os exercícios **NÃO SÃO OPCIONAIS**. O objetivo de cada exercício deste caderno é a fixação de determinado conceito. É aí que reside a importância da realização de todos. Sugerimos fortemente que, em caso de dúvida, em algum exercício você entre em contato com seu tutor externo ou com a tutoria da UNIASSELVI e que não passe para o exercício seguinte enquanto o atual não estiver completamente compreendido.

Por fim, ressalto que mesmo sendo uma área muito ampla, o Caderno de Estudos lhe oferece um início sólido e consistente sobre o tema. Desejo a você uma excelente experiência nos estudos dos conteúdos dessa disciplina!

Prof. Pedro Sidnei Zanchett, MEgc.



Você já me conhece das outras disciplinas? Não? É calouro? Enfim, tanto para você que está chegando agora à UNIASSELVI quanto para você que já é veterano, há novidades em nosso material.

Na Educação a Distância, o livro impresso, entregue a todos os acadêmicos desde 2005, é o material base da disciplina. A partir de 2017, nossos livros estão de visual novo, com um formato mais prático, que cabe na bolsa e facilita a leitura.

O conteúdo continua na íntegra, mas a estrutura interna foi aperfeiçoada com nova diagramação no texto, aproveitando ao máximo o espaço da página, o que também contribui para diminuir a extração de árvores para produção de folhas de papel, por exemplo.

Assim, a UNIASSELVI, preocupando-se com o impacto de nossas ações sobre o ambiente, apresenta também este livro no formato digital. Assim, você, acadêmico, tem a possibilidade de estudá-lo com versatilidade nas telas do celular, tablet ou computador.

Eu mesmo, UNI, ganhei um novo *layout*, você verá frequentemente e surgirei para apresentar dicas de vídeos e outras fontes de conhecimento que complementam o assunto em questão.

Todos esses ajustes foram pensados a partir de relatos que recebemos nas pesquisas institucionais sobre os materiais impressos, para que você, nossa maior prioridade, possa continuar seus estudos com um material de qualidade.

Aproveito o momento para convidá-lo para um bate-papo sobre o Exame Nacional de Desempenho de Estudantes – ENADE.

Bons estudos!

BATE SOBRE O PAPO ENADE!



Olá, acadêmico!



Você já ouviu falar sobre o ENADE?

Se ainda não ouviu falar nada sobre o ENADE, agora você receberá algumas informações sobre o tema.

Ouviu falar? Ótimo, este informativo reforçará o que você já sabe e poderá lhe trazer novidades.



Vamos lá!



Qual é o significado da expressão ENADE?

EXAME NACIONAL DE DESEMPENHO DOS ESTUDANTES

Em algum momento de sua vida acadêmica você precisará fazer a prova ENADE.



Que prova é essa?

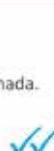
É **obrigatória**, organizada pelo INEP – Instituto Nacional de Estudos e Pesquisas Educacionais Anísio Teixeira.

Quem determina que esta prova é obrigatória... O **MEC** – Ministério da Educação.

O objetivo do MEC com esta prova é o de avaliar seu desempenho acadêmico assim como a qualidade do seu curso.



Fique atento! Quem não participa da prova fica impedido de se formar e não pode retirar o diploma de conclusão do curso até regularizar sua situação junto ao MEC.



Não se preocupe porque a partir de hoje nós estaremos auxiliando você nesta caminhada.



Você receberá outros informativos como este, complementando as orientações e esclarecendo suas dúvidas.



Participará de webconferências entre outras tantas atividades para que esteja preparado para #mandar bem na prova ENADE.

Nós aqui no NEAD e também a equipe no polo estamos com você para vencermos este desafio.

Conte sempre com a gente, para juntos mandarmos bem no ENADE!



SUMÁRIO

UNIDADE 1 - EVOLUÇÃO DO SOFTWARE, FUNDAMENTOS DE ENGENHARIA DE SOFTWARE, CICLO DE VIDA DE DESENVOLVIMENTO DE SOFTWARE E REQUISITO DE SOFTWARE	1
TÓPICO 1 - EVOLUÇÃO DO SOFTWARE	3
1 INTRODUÇÃO	3
2 EVOLUÇÃO DO SOFTWARE	7
3 TIPOS DE SOFTWARE DO PONTO DE VISTA DA ENGENHARIA	9
RESUMO DO TÓPICO 1.....	14
AUTOATIVIDADE	16
TÓPICO 2 - FUNDAMENTOS DE ENGENHARIA DE SOFTWARE	19
1 INTRODUÇÃO	19
2 PRINCÍPIOS DA ENGENHARIA DE SOFTWARE	22
3 METODOLOGIA DE DESENVOLVIMENTO DE SISTEMAS	23
3.1 FASES DE DESENVOLVIMENTO DE SOFTWARE	27
3.1.1 Fase de iniciação	28
3.1.2 Fase de elaboração	29
3.1.3 Fase de construção	29
3.1.4 Fase de transição	29
4 PROCESSOS DE ENGENHARIA DE SOFTWARE.....	30
LEITURA COMPLEMENTAR.....	33
RESUMO DO TÓPICO 2.....	39
AUTOATIVIDADE	41
TÓPICO 3 – CICLO DE VIDA DE DESENVOLVIMENTO DE SOFTWARE	45
1 INTRODUÇÃO	45
2 MODELOS DE PROCESSO DE CICLO DE VIDA DE SOFTWARE.....	46
2.1 MODELO CASCATA OU SEQUENCIAL	47
2.2 MODELOS POR PROTOTIPAÇÃO	48
2.3 MODELO ESPIRAL	50
2.4 MODELO ITERATIVO E INCREMENTAL	52
2.5 MODELO BASEADO EM COMPONENTES	54
2.6 MODELO EM V	56
2.7 MODELO RAD (RAPID APPLICATION DEVELOPMENT).....	57
2.8 MODELO DE QUARTA GERAÇÃO	58
RESUMO DO TÓPICO 3.....	60
AUTOATIVIDADE	62
TÓPICO 4 – REQUISITO DE SOFTWARE	65
1 INTRODUÇÃO	65
1.1 REQUISITOS FUNCIONAIS	68
1.2 REQUISITOS NÃO FUNCIONAIS.....	68
1.3 REQUISITOS INVERSOS	68

2 TÉCNICAS DE LEVANTAMENTO DE REQUISITOS.....	69
3 DEFINIÇÕES E PRÁTICAS DE GERÊNCIA DE REQUISITOS PELAS PRINCIPAIS NORMAS DE DESENVOLVIMENTO DE SOFTWARE.....	72
3.1 CMMI	72
3.1.1 Gerenciar requisitos.....	72
3.1.1.1 Obter um entendimento dos requisitos.....	72
3.1.1.2 Gerenciar mudanças de requisitos	73
3.1.1.3 Manter rastreabilidade bidirecional dos requisitos	73
3.1.1.4 Encontrar inconsistências entre trabalho de projeto e requisitos	73
3.2 MPS.BR	74
4 GERENCIAR MUDANÇAS DE REQUISITOS.....	75
RESUMO DO TÓPICO 4.....	77
AUTOATIVIDADE	79
UNIDADE 2 - GERENCIAMENTO DE PROJETOS DE SOFTWARE, ESTIMATIVAS E MÉTRICAS DE PROJETOS DE SOFTWARE E GERÊNCIA DE CONFIGURAÇÃO DE SOFTWARE.....	81
TÓPICO 1 - GERENCIAMENTO DE PROJETOS DE SOFTWARE	83
1 INTRODUÇÃO	83
2 DEFINIÇÃO DE PROJETO	83
3 GERÊNCIA DE PROJETOS.....	85
4 SUBPROJETOS, PROGRAMAS E PORTFÓLIO	86
5 FASES DA GERÊNCIA DE PROJETOS	88
6 ESTRUTURA ORGANIZACIONAL	89
7 GERÊNCIA DE RISCOS	92
8 GESTÃO DE PROJETOS: PMI E O PMBOK	94
9 GESTÃO DE PESSOAS EM PROJETOS DE SOFTWARE.....	99
10 FERRAMENTAS PARA GERÊNCIA DE PROJETOS	103
10.1 MS PROJECT	103
10.2 GANTT PROJECT	104
10.3 DOTPROJECT	105
10.4 PROJECT OPEN	106
RESUMO DO TÓPICO 1.....	108
AUTOATIVIDADE	111
TÓPICO 2 - ESTIMATIVAS E MÉTRICAS DE PROJETOS DE SOFTWARE.....	117
1 INTRODUÇÃO	117
2 O GERENCIAMENTO DE CUSTOS: GUIA PMBOK	118
3 MÉTRICAS DE SOFTWARE.....	119
4 MÉTRICAS UTILIZADAS PARA ESTIMAR SISTEMAS	120
4.1 LINHA DE CÓDIGO (LOC).....	120
4.2 PONTOS DE HISTÓRIA	121
4.3 ANÁLISE DE PONTOS DE FUNÇÃO (APF).....	122
4.4 PONTOS DE CASO DE USO (PCU)	127
4.5 MODELO COCOMO II	130
4.6 ESTIMATIVA PARA PROJETOS ORIENTADOS A OBJETO	132
LEITURA COMPLEMENTAR.....	133
RESUMO DO TÓPICO 2.....	140
AUTOATIVIDADE	143
TÓPICO 3 - GERÊNCIA DE CONFIGURAÇÃO DE SOFTWARE	145
1 INTRODUÇÃO	145
2 ITENS DE CONFIGURAÇÃO	146

3 O AUXÍLIO DA GERÊNCIA DE CONFIGURAÇÃO DE SOFTWARE.....	147
4 AUTORIA DE CONFIGURAÇÃO	151
5 RELATÓRIO DE STATUS DE CONFIGURAÇÃO.....	152
6 TERMINOLOGIAS DA GERÊNCIA DE CONFIGURAÇÃO DE SOFTWARE	152
RESUMO DO TÓPICO 3.....	155
AUTOATIVIDADE	157

UNIDADE 3 - GERENCIAMENTO DE QUALIDADE DE SOFTWARE: PADRÕES, NORMAS E MODELOS; MÉTODOS ÁGEIS; VERIFICAÇÃO, VALIDAÇÃO E TESTES DE SOFTWARE; GOVERNANÇA DE TECNOLOGIA DA INFORMAÇÃO **161**

TÓPICO 1 - GERENCIAMENTO DE QUALIDADE DE SOFTWARE	
PADRÕES, NORMAS E MODELOS	163
1 INTRODUÇÃO	163
2 ABORDAGENS DA QUALIDADE.....	164
3 TOTAL QUALITY MANAGEMENT (TQM)	166
4 INTRODUÇÃO À QUALIDADE DE SOFTWARE	167
4.1 GARANTIA E CONTROLE DA QUALIDADE DE SOFTWARE	168
5 PADRÕES, NORMAS E MODELOS DE QUALIDADE DE SOFTWARE.....	170
5.1 NORMA ISO/IEC 9000.....	171
5.2 NORMA ISO/IEC 12207.....	172
5.3 NORMA ISO/IEC 15504.....	173
5.4 NORMA ISO/IEC ISO 9126.....	175
5.5 NORMA ISO/IEC 27000.....	177
5.6 NORMA ISO/IEC 15939.....	178
6 MODELOS CMMI E MPS.BR	178
6.1 CMMI (CAPABILITY MATURITY MODEL INTEGRATION): INTEGRAÇÃO DOS MODELOS DE CAPACITAÇÃO E MATURIDADE DE SISTEMAS	179
6.2 MELHORIA DE PROCESSO DE SOFTWARE BRASILEIRO (MPS.BR)	182
6.3 COMPARAÇÃO CMMI E MPS.BR	185
RESUMO DO TÓPICO 1.....	187
AUTOATIVIDADE	191
TÓPICO 2 - MÉTODOS ÁGEIS.....	195
1 INTRODUÇÃO	195
2 CARACTERÍSTICAS DOS MÉTODOS ÁGEIS.....	197
3 PRINCIPAIS MÉTODOS ÁGEIS	198
3.1 SCRUM	199
3.2 EXTREME PROGRAMMING	201
3.3 ADAPTATIVE SOFTWARE DEVELOPMENT (ASD).....	203
3.4 DYNAMIC SYSTEM DEVELOPMENT METHOD (DSDM).....	204
3.5 CRYSTAL CLEAR	205
3.6 FEATURE-DRIVEN DEVELOPMENT (FDD).....	205
4 BENEFÍCIOS E MALEFÍCIOS DA METODOLOGIA ÁGIL.....	207
RESUMO DO TÓPICO 2.....	209
AUTOATIVIDADE	212
TÓPICO 3 - VERIFICAÇÃO, VALIDAÇÃO E TESTES DE SOFTWARE	215
1 INTRODUÇÃO	215
1.1 VALIDAÇÃO	216
1.2 VERIFICAÇÃO	216

1.3 TESTE	217
2 EQUIPE DE TESTE.....	219
3 ERROS DE SOFTWARE.....	220
4 TIPOS DE TESTE.....	221
4.1 FUNCIONALIDADE	222
4.2 USABILIDADE	222
4.3 CONFIABILIDADE.....	222
4.4 DESEMPENHO	222
4.5 SUPORTABILIDADE	223
5 PROCESSO DE TESTE DE SOFTWARE.....	223
5.1 PRÁTICAS DE DESENVOLVIMENTO	224
5.1.1 TDD - <i>Test-Driven Development</i> (Desenvolvimento Guiado a Testes).....	224
5.1.2 DDD - <i>Domain-Driven Design</i> (Desenvolvimento Guiado ao Domínio)	224
5.1.3 BDD – <i>Behavior-Driven Development</i> (Desenvolvimento Guiado Por Comportamento)	225
5.1.4 ATDD - <i>Acceptance Test-Driven Development</i> (Desenvolvimento Guiado por Testes de Aceitação).....	225
5.1.5 FDD - <i>Feature Driven Development</i> (Desenvolvimento Guiado por Funcionalidades)	226
6 FERRAMENTAS PARA AUTOMAÇÃO DE TESTE	226
RESUMO DO TÓPICO 3.....	229
AUTOATIVIDADE	232
 TÓPICO 4 - GOVERNANÇA DE TECNOLOGIA DA INFORMAÇÃO.....	235
1 INTRODUÇÃO	235
2 CONTROL OBJECTIVES FOR INFORMATION AND RELATED TECHNOLOGY (COBIT)	237
3 INFORMATION TECHNOLOGY INFRASTRUCTURE LIBRARY (ITIL).....	239
3.1 PROVEDOR DE SERVIÇOS DE TI.....	243
3.2 TIPOS DE SERVIÇOS DE TI.....	244
3.3 PROCESSOS E FUNÇÕES	244
3.4 PAPÉIS	245
3.5 MATRIZ RACI (<i>RESPONSIBLE, ACCOUNTABLE, CONSULTED, INFORMED</i>)	246
LEITURA COMPLEMENTAR.....	246
RESUMO DO TÓPICO 4.....	250
AUTOATIVIDADE	253
REFERÊNCIAS	255

UNIDADE 1

EVOLUÇÃO DO SOFTWARE, FUNDAMENTOS DE ENGENHARIA DE SOFTWARE, CICLO DE VIDA DE DESENVOLVIMENTO DE SOFTWARE E REQUISITO DE SOFTWARE

OBJETIVOS DE APRENDIZAGEM

Ao final desta unidade você será capaz de:

- compreender os principais conceitos da evolução do *software*;
- entender a área de Engenharia de *Software* de maneira sistêmica;
- compreender as principais metodologias de ciclo de vida de *software* e processos de *software*;
- conhecer a importância de requisitos de *software*.

PLANO DE ESTUDOS

Esta unidade de ensino contém quatro tópicos. No final de cada um deles você encontrará atividades que contribuirão para a apropriação dos conteúdos.

TÓPICO 1 – EVOLUÇÃO DO SOFTWARE

TÓPICO 2 – FUNDAMENTOS DE ENGENHARIA DE SOFTWARE

TÓPICO 3 – CICLO DE VIDA DE DESENVOLVIMENTO DE SOFTWARE

TÓPICO 4 – REQUISITO DE SOFTWARE

EVOLUÇÃO DO SOFTWARE

1 INTRODUÇÃO

A evolução do *software* confunde-se com a evolução dos computadores, inicialmente compostos apenas do *hardware*, onde toda a lógica de processamento era executada no meio físico. Porém, à medida que o *hardware* evolui, o *software* também acompanha essa mudança devido à necessidade de se tornar acessível ao usuário final.

O computador necessita que o *software* e o *hardware* andem em paralelo, para que haja um melhor aproveitamento dos recursos. De nada adianta o melhor *hardware* do mundo usando um sistema operacional defasado, tanto como um aplicativo de última geração tentar rodar em um *hardware* antigo e desatualizado.

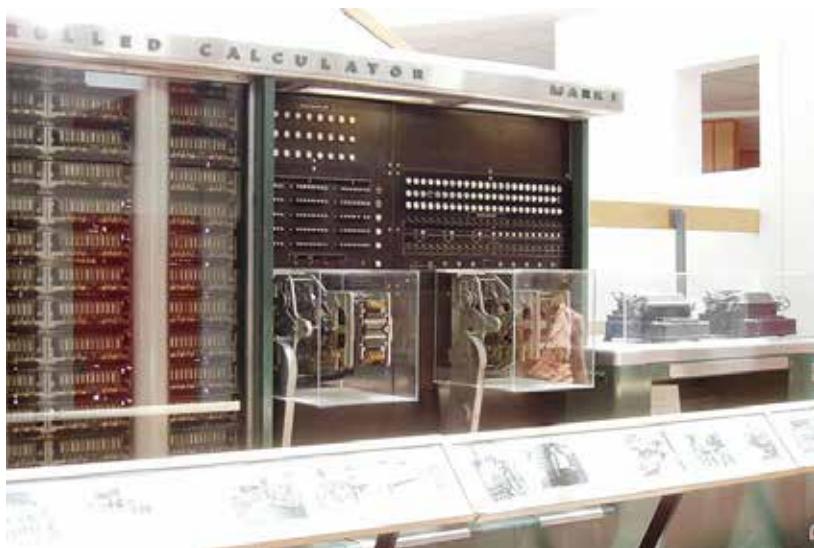
A evolução do *software* passa, também, pelas características históricas do *hardware*. Roger Pressman (2009), guru da Engenharia de *Software*, define o *software* como componentes não executáveis em máquina e componentes executáveis em máquina. Explica-nos que o *software* é criado por meio de uma série de regras que mapeiam as exigências de negócios que são desenvolvidos em linguagem de programação, que especifica a estrutura de seus dados, os atributos procedimentais e os requisitos relacionados para código e os converte em instruções executáveis em máquina.

O *software* engloba códigos que executam funções entre si e, normalmente, retornam alguma informação ao usuário. Para a criação do *software*, os programadores utilizam uma linguagem de programação, que é interpretada por um compilador, que a transforma para código binário, o qual é lido pelo *hardware*.

Para entendermos essa definição, segue um pouco da história da “era do computador”.

O primeiro computador surgiu para fins militares na década de 1940, o “Mark I” (tradução por tanque de guerra), foi financiado pela Marinha norte-americana em conjunto com a Universidade de Harvard e a IBM, onde ocupava, aproximadamente, 120 m³ e tinha 4,5 toneladas. Um gigante eletromagnético lançado no ano de 1944 para auxiliar nos cálculos de precisão necessários para balística (BERNARDES, 2015).

FIGURA 1 - MARK I

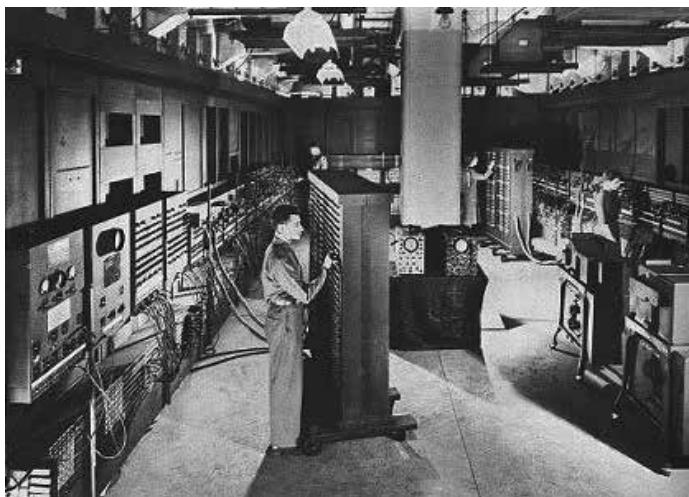


FONTE: Disponível em: <<http://www.geocities.ws/trabalhoweb2/>>. Acesso em: 03 jul. 2015

Nessa época, devido às limitações da “linguagem máquina”, as operações para realizar os processos eram complexas, demandando várias pessoas para manter o funcionamento dos computadores. No período, uma das criações mais notáveis – influenciando, inclusive, na Segunda Guerra Mundial – foi a máquina de Alan Turing, que foi capaz de decifrar os códigos utilizados pela Alemanha. Turing é considerado uma das maiores mentes na Ciência da Computação, tendo vários conceitos sendo estudados ainda nos dias atuais, como a Inteligência Artificial (IA).

Na mesma década, em 1946, surgiu o primeiro computador eletrônico à válvula (relés eletromagnéticos e máquinas perfuradoras de cartões), desenvolvido por Eckert e Mauchly, o “ENIAC” (*Electronic Numerical Integrator and Computer* ou Computador Integrador Numérico Eletrônico), cuja velocidade de processamento era superior à do Mark I e configurações mais próximas dos computadores atuais. Ocupava, aproximadamente, 180 m² e tinha 30 toneladas. Sua memória era muito pequena, e a cada nova operação era necessário reconfigurar toda a sua fiação, exigindo um enorme esforço humano.

FIGURA 2 - ENIAC

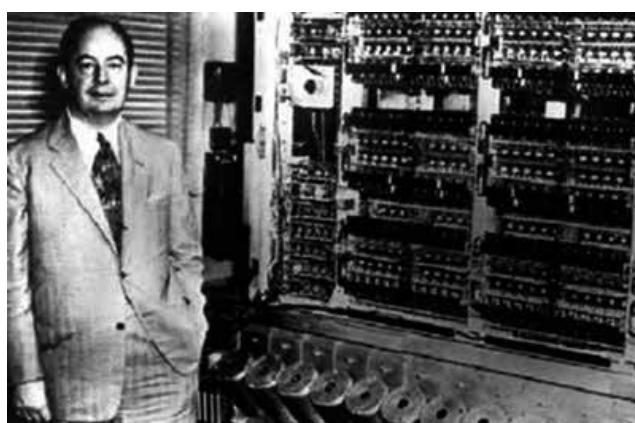


FONTE: Disponível em: <<http://www.geocities.ws/trabalhoweb2/>>. Acesso em: 03 jul. 2015

A principal mudança da primeira geração para a segunda foi a substituição da válvula para o transistor, deixando-se de utilizar a linguagem de máquina para a programação em Assembly, facilitada por dispositivos de entrada e saída, melhorando a comunicação homem-máquina.

Portanto, o primeiro computador eletrônico com programa armazenado foi o “EDVAC” (*Electronic Discrete Variable Automatic Computer* ou Computador Eletrônico com Discreta Variação Automática), desenvolvido por Von Neumann. Utilizava o sistema binário, uma máquina digital de armazenamento (“memória”) para comportar, respectivamente, instruções e dados na mesma unidade de processamento (CPU), podendo, assim, manipular tais programas. Tornou-se a arquitetura padrão para os computadores mais modernos, pois era dotado de cem vezes mais memória interna do que o ENIAC, um grande salto para a época, e ocupava 45,5 m² com quase oito toneladas (BERNARDES, 2015).

FIGURA 3 - EDVAC



FONTE: Disponível em: <<http://www.geocities.ws/trabalhoweb2/>>. Acesso em: 03 jul. 2015

Em 1949, Maurice Wilkes criou o EDSAC (*Electronic Delay Storage Automatic Calculator* ou Calculadora Automática com Armazenamento por Retardo Eletrônico), o primeiro computador operacional em grande escala capaz de armazenar seus próprios programas.

Na década de 50 surgiu o primeiro computador comercial do mundo, o LEO (Escritório Eletrônico de Lyons), que começou a automatizar os trabalhos de escritórios, criado através da parceria entre a J. Lyons e a Universidade de Cambridge e começou a ser utilizado em 1954.

FIGURA 4 - LEO



FONTE: Disponível em: <<http://www.geocities.ws/trabalhoweb2/>>. Acesso em: 03 jul. 2015

A partir de 1953, o uso de circuitos integrados diminuiu o tamanho dos computadores e a empresa IBM começou a comercializar o computador IBM 701, onde eram utilizados cartões perfurados para o armazenamento de programas e dados.

FIGURA 5 - CONSOLE DO IBM 701



FONTE: Disponível em: <<http://www.geocities.ws/trabalhoweb2/>>. Acesso em: 03 jul. 2015

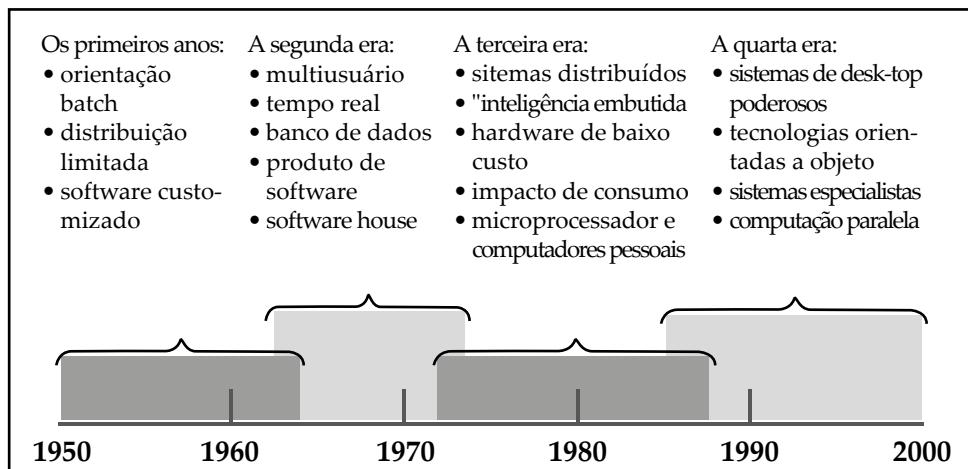
No final dos anos 60, o acesso a informações ficou mais rápido, foram introduzidas linguagens de alto nível, como Cobol e Fortran, complementadas por sistemas de armazenamento em disco e fitas magnéticas. Com o surgimento das linguagens imperativas Cobol e Fortran, as possibilidades de programação se expandiram vastamente.

O Cobol ainda é utilizado amplamente nos dias atuais em ambientes de *mainframe*, principalmente em bancos, devido à sua estabilidade, recebendo até expansão para ser utilizado como linguagem orientada a objeto. Outro ponto é o custo para realizar a migração e integração de dados para uma plataforma mais atual, que é muito alto.

2 EVOLUÇÃO DO SOFTWARE

Conforme visto, o desenvolvimento do *software* está estreitamente ligado há mais de cinco décadas de evolução do *hardware*, culminando em menor tamanho do *hardware*, fazendo com que sistemas baseados em computadores se tornassem mais sofisticados. Evoluímos dos processadores à válvula para os dispositivos microeletrônicos, que são capazes de processar milhares de instruções por segundo. A figura a seguir descreve a evolução do *software* dentro do contexto das áreas de aplicação de sistemas baseados em computador.

FIGURA 6 – A EVOLUÇÃO DO SOFTWARE



FONTE: Disponível em: <<http://www.batebyte.pr.gov.br/modules/conteudo/conteudo.php?conteudo=299>>. Acesso em: 03 de jul. 2015.

Conforme destacado por Azevedo (2015), a evolução do *software* possui as seguintes características:

- Os primeiros anos:** como já visto anteriormente, no início, o *hardware* sofreu contínuas mudanças e o *software* foi incorporando-se aos poucos nesta evolução, na qual o *hardware* dedicava-se à execução de um único programa que, por sua vez, dedicava-se a uma única aplicação específica. Tinha-se uma

distribuição limitada e com poucos métodos sistemáticos. Como característica usava uma orientação *batch* (em lote) para a maioria dos sistemas. A maior parte dos *softwares* era desenvolvida e usada pela própria empresa e quem escrevia e colocava em funcionamento também tratava dos defeitos, e por conta do ambiente de *software* personalizado, o projeto era um processo implícito, realizado no cérebro de alguém, e a documentação muitas vezes não existia.

- **A segunda era:** entre a década de 1960 até o final da década de 1970 foi o período da multiprogramação. Os sistemas multiusuários ofereceram sofisticação de *software* e *hardware*, melhorando a interação homem-máquina. Houve o surgimento de sistemas de tempo real que podiam coletar, analisar e transformar dados de múltiplas fontes. Os avanços da armazenagem *online* levaram à primeira geração de sistemas de gerenciamento de banco de dados e surgimento dos *softwares houses*. Os programas para *mainframes* e minicomputadores eram distribuídos para centenas e, às vezes, milhares de usuários, e à medida que cresciam, devido à natureza personalizada de muitos programas, tornava-os virtualmente impossíveis de sofrerem manutenção. Uma "crise de *software*" agigantou-se no horizonte.



Os conceitos e a origem da engenharia de *software*, a fim de eliminar esta crise, bem como suas características, poderão ser vistos na página 15, Tópico 2, desta unidade de ensino.

- **A terceira era:** começou em meados da década de 1970 e continua até hoje. Existência dos sistemas distribuídos e múltiplos computadores, onde cada um, executando funções concorrentemente e comunicando-se um com o outro, aumenta intensamente a complexidade dos sistemas baseados em computador. As redes globais, as comunicações digitais de largura de banda elevada e a crescente demanda de acesso "instantâneo" a dados exigem muito dos desenvolvedores de *software*. Foi caracterizada, também, pelo advento e o generalizado uso de microprocessadores, computadores pessoais e poderosas estações de trabalho "*workstations*" de mesa. O microprocessador gerou um amplo conjunto de produtos inteligentes. Do automóvel a fornos micro-ondas, de robôs industriais a equipamentos para diagnóstico de soro sanguíneo. Em muitos casos, a tecnologia de *software* está sendo integrada a produtos, por equipes técnicas que entendem de *hardware*, mas que frequentemente são principiantes em desenvolvimento de *software*. O *hardware* de computador pessoal está se tornando rapidamente um produto primário, enquanto o *software* oferece a característica capaz de diferenciar.

- **A quarta era:** esta era está apenas começando. As tecnologias orientadas a objetos, orientadas a documentos, estão ocupando o lugar das abordagens mais convencionais para o desenvolvimento de *software* em muitas áreas de aplicação. As técnicas de "quarta geração" para o desenvolvimento de *software* já estão mudando a maneira segundo a qual alguns segmentos da comunidade de *software* constroem programas de computador. Os sistemas especialistas e o *software* de inteligência artificial finalmente saíram do laboratório para a aplicação prática em problemas de amplo espectro do mundo real. O *software* de rede neural artificial abriu excitantes possibilidades para o reconhecimento de padrões e para capacidades de processamento de informações semelhantes às humanas.
- **Era atual:** a sofisticação do *software* ultrapassou nossa capacidade de construir um *software* que extraia o potencial do *hardware*. A capacidade de construir programas não acompanha o ritmo da demanda de novos programas, a capacidade de manter os programas existentes é ameaçada por projetos ruins e recursos inadequados e as práticas de engenharia de *software* se fortalecem.

3 TIPOS DE SOFTWARE DO PONTO DE VISTA DA ENGENHARIA

O *software* é um conjunto de algoritmos codificados que permite ao computador executar uma operação ou um conjunto de operações culminando em tarefas. Analisaremos, aqui, os tipos de *software* disponíveis, bem como a função e utilidade desses tipos de *software*.

Segundo Azevedo (apud VERZELLO, 1984), o *software* é classificado em três tipos: (1) *Software* de sistema, que são programas escritos para controlar e coordenar o *software*; (2) *Software* de linguagens, que são programas que traduzem outros programas para a forma binária, que é a linguagem utilizada pelos componentes do sistema computacional para mantê-los salvos em bancos de dados especiais; e, por último, (3) *Software* de aplicação, que são programas escritos para resolver problemas comerciais ou prestar outros serviços de processamento de dados aos usuários.

Roger Pressman (1995), porém, amplia essa classificação de *software* em sete diferentes categorias, afirmando ser uma tarefa um tanto difícil desenvolver categorias genéricas para aplicações de *softwares*, pois, no mesmo passo que o *software* cresce, desaparece a visão de compartimentos:

1. ***Software* Básico:** é uma coleção de programas que dão apoio a outros programas. É caracterizado pela forte interação com *hardware*, intenso uso por múltiplos usuários; operações concorrentes que exigem escalonamento *schedule*; compartilhamento de recursos e sofisticada administração do processo; estruturas de dados complexas e múltiplas interfaces externas. Exemplo: compiladores, editores simples, *drivers*, componentes do SO.
2. ***Software* de Tempo Real:** monitorar, analisar e controlar eventos do mundo real, caracterizado pela coleta de dados do ambiente externo, análise que transforma a informação de acordo com a necessidade do sistema, controle e saída para um ambiente externo e um componente de monitoração que

coordena todos os outros. O termo tempo real difere de interativo ou tempo compartilhado, pois deve responder dentro de restrições de tempos exatos sem resultados desastrosos.

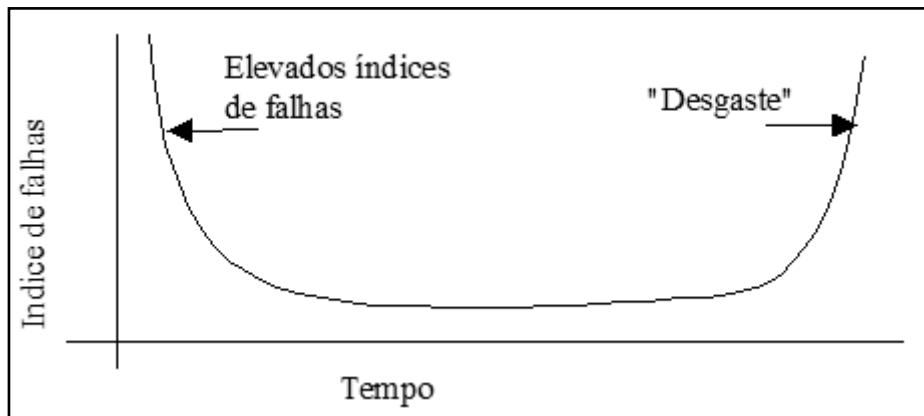
3. **Software Comercial:** facilita as operações comerciais e decisões administrativas. As aplicações dessa área reestruturam os dados de uma forma que facilita as operações comerciais e as tomadas de decisões administrativas. Além da aplicação de processamento de dados convencional, as aplicações de *software* comerciais abrangem a computação interativa. Exemplos: controle de estoque, finanças, vendas etc.
4. **Software Científico e de Engenharia:** algoritmos com intenso processamento de números e cálculos. As aplicações diversificadas variam da análise de fadiga mecânica de automóveis à dinâmica orbital de naves espaciais recuperáveis, e da biologia molecular à manufatura automatizada. Exemplos: sistemas de astronomia, naves espaciais, matemática avançada etc.
5. **Software Embutido:** é usado para controlar produtos e sistemas para mercados industriais e de consumo, pode utilizar memória de somente leitura e usa rotinas limitadas e particulares. O *software* embutido reside na memória só de leitura (*read only*) e pode executar funções limitadas e particulares (por exemplo, controle de teclado para fornos de micro-ondas) ou oferecer recursos funcionais de controle significativos (por exemplo, funções digitais em automóveis, tais como controle, mostradores no painel, sistemas de freio etc.).
6. **Software de computador pessoal:** utilizados em computadores de uso pessoal. Exemplos: editores de texto, planilhas, calculadora, jogos, computação gráfica, gerenciamento de dados, aplicações financeiras pessoais e comerciais, redes externas ou acesso a banco de dados, são apenas algumas das centenas de aplicações.
7. **Software de inteligência artificial:** faz uso de algoritmos não numéricos para resolver problemas complexos que não sejam favoráveis à computação ou à análise direta. Atualmente, a área de *Artificial Intelligence* (AI) mais ativa é a dos sistemas especialistas baseados em conhecimentos, porém outras áreas de aplicação para o *software* de AI são o reconhecimento de padrões (voz e imagem), jogos e demonstração de teoremas. Uma rede neural simula a estrutura dos processos cerebrais (a função do neurônio biológico) e pode levar a uma nova classe de *software* que consegue reconhecer padrões complexos e aprender com a experiência passada. Exemplos: sistema de reconhecimento de imagem, sistemas especialistas, redes neurais e aprendizado etc.

Diante do contexto histórico da evolução do computador, podemos perceber que o *software* é um elemento de sistema lógico e complexo, projetado pela engenharia, e o *hardware*, um elemento físico projetado por manufatura. Ambos apresentam características diferentes quanto à forma de utilização. Por exemplo: com o passar dos tempos, o *hardware* vai desgastando-se, exigindo manutenção, e o *software*, por sua vez, deteriora-se, exigindo sua evolução (vide Figura 7).

Os índices de falhas são relativamente elevados logo no começo do ciclo de vida do *hardware*, os defeitos são corrigidos e o índice de falhas cai para estável,

porém, à medida que o tempo passa, o índice de falhas eleva-se novamente, conforme os componentes de *hardware* vão sofrendo males ambientais (poeira, vibração, abuso, temperaturas extremas etc.).

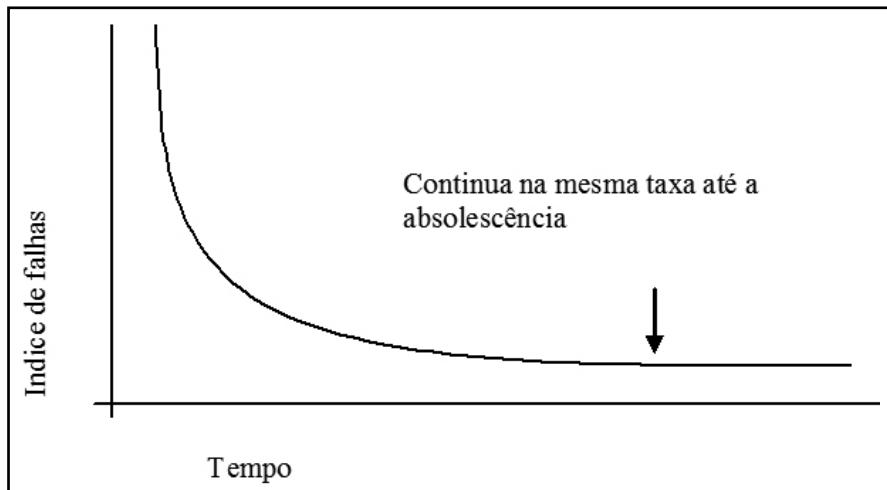
FIGURA 7 - CURVAS DE FALHAS PARA O HARDWARE



FONTE: Disponível em: <<http://blog.tecsystem.com.br/index.php/erros-e-acertos-no-desenvolvimento-de-software/>>. Acesso em: 05 jul. 2015.

Já o *software* não é sensível aos problemas ambientais. Os defeitos não descobertos no começo da vida de um programa provocarão elevados índices de falhas; após corrigidas, a curva achata-se; entretanto, fica claro que o *software* não se desgasta. Todavia, se deteriora!

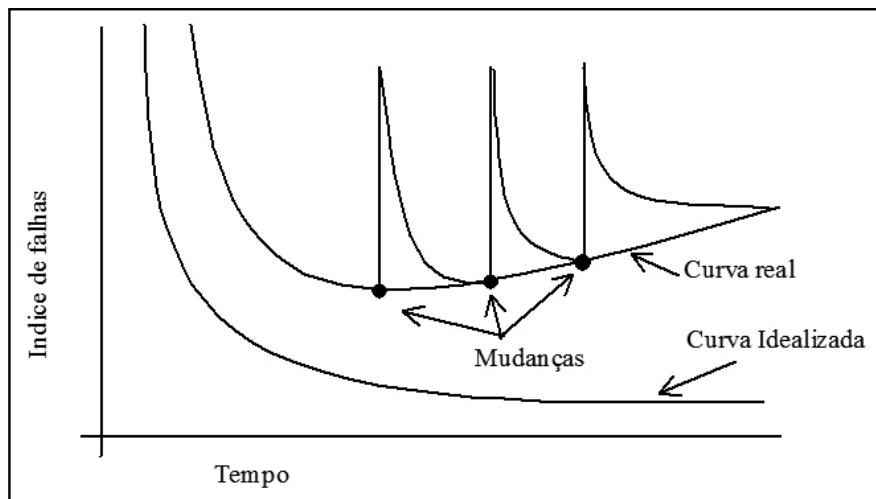
FIGURA 8 - CURVA DE FALHAS DO SOFTWARE (IDEALIZADA)



FONTE: Disponível em: <<http://blog.tecsystem.com.br/index.php/erros-e-acertos-no-desenvolvimento-de-software/>>. Acesso em: 05 jul. 2015.

Portanto, segundo Amparo (2015), esta aparente contradição pode ser mais bem explicada considerando a Figura 9. Durante sua vida, o *software* enfrentará mudanças (manutenção). Quando estas são feitas, é provável que novos defeitos sejam introduzidos, fazendo com que a curva do índice de falhas apresente picos.

FIGURA 9 - CURVA DE FALHAS REAL PARA O SOFTWARE



FONTE: Disponível em: <<http://blog.tecsystem.com.br/index.php/erros-e-acertos-no-desenvolvimento-de-software/>>. Acesso em: 05 jul. 2015

Percebe-se que a manutenção do *software* envolve consideravelmente mais complexidade do que a de *hardware*. Basta substituir um componente de *hardware* por uma peça de reposição; já com o *software* não existe reposição de peças, toda falha indica um erro de projeto ou no processo por meio do qual o projeto é especificado e traduzido em código executável por máquina.

É necessário antecipar os caminhos em que o *software* sofre mudanças, dessa forma pode-se modificá-lo mais facilmente para acomodar tais necessidades. Porém, antecipar-se às mudanças não é uma tarefa fácil, uma vez que existem muitas razões pelas quais os sistemas mudam.



Dante destes fatores, a engenharia de *software* vem para justificar que as principais causas estão na introdução de erros no processo de execução dos projetos de *software*. Ocorrem devido a projetos com má especificação, mal planejados, sem treinamentos, má implementação, testes incompletos ou mal feitos e, por fim, problemas na comunicação homem-máquina.

Furtado (2007) conclui que, com o passar do tempo, os *softwares* vão “envelhecendo”, sua estrutura se torna ultrapassada e, por já não satisfazerem suas tarefas como antes, são descartados. O *software* é inherentemente flexível e pode ser alterado. Raramente é necessário descartar um *software* inteiro por ele

não ser capaz de realizar determinada tarefa ou atingir um novo resultado. Em determinadas circunstâncias é preferível adaptar o *software* ao invés de eliminá-lo totalmente. Porém, para que isso seja possível é necessário que sejam adotadas técnicas de evolução de *software*, dessa forma o processo de reestruturação de um *software* pode ser possível, evitando que ele perca sua utilidade.

Estas técnicas de evolução devem auxiliar o *software* a não sofrer tantos impactos após sucessivas versões. Como as modificações estão presentes na vida de um *software*, faz-se necessário utilizar controles a fim de reverter o envelhecimento do *software*, ou seja, adiar o dia em que este já não poderá ser mais viável.

RESUMO DO TÓPICO 1

Neste tópico, você aprendeu que:

- O *hardware* e o *software* sempre andaram em paralelo, para um funcionar precisava da evolução dos recursos do outro. O *software* é definido como componentes não executáveis em máquina e componentes executáveis em máquina.
- A história do computador se iniciou em 1940, com o Mark I, feito para fins militares; em seguida, em 1946, surgiu o primeiro computador à válvula, o ENIAC (Computador Integrador Numérico Eletrônico), que tinha mais velocidade de execução. Na mesma década surgiram, também, o primeiro computador eletrônico com programa armazenado, o EDVAC (Computador Eletrônico com Discreta Variação Automática) e o primeiro computador operacional, o EDSAC (Calculadora Automática com Armazenamento por Retardo Eletrônico). Por fim, na década de 50, o computador comercial, o LEO (Escritório Eletrônico de Lyons), denominado IBM 701.
- Conforme já visto, o *software* veio para controlar o *hardware* e ajudar o usuário em suas tarefas. Inicialmente, preso a tarefas mais industriais até chegar no uso do Computador Pessoal.
- De modo geral, as áreas de aplicações da evolução do *software* são:
 - Os primeiros anos: a partir de 1950 eram orientados a *batch*, distribuição limitada e *software* customizado.
 - A segunda era: a partir de 1960, o *software* multiusuário, tempo real, banco de dados, produção de *software* e *software house*.
 - A terceira era: a partir de 1970, os sistemas distribuídos, *hardware* de baixo custo, microprocessadores e computadores pessoais.
 - A quarta geração: a partir de 1980, os sistemas *desktop* poderosos, tecnologia orientada a objeto, sistemas especialistas e computação paralela.
- Robert J. Verzello classifica o *software* pelos tipos: *software* de sistemas, *software* de linguagens e *software* de aplicações. Já Roger Pressman (2002) o classifica pelos tipos: *software* básico, *software* de tempo real, *software* comercial, *software* científico e de engenharia, *software* embutido, *software* de computador pessoal e *software* de inteligência artificial.

- Finalizando, o *software* é um elemento de sistema lógico e complexo projetado pela engenharia e o *hardware* um elemento físico projetado por manufatura. Com o passar dos tempos, o *hardware* foi desgastando-se devido ao ambiente físico, exigindo a manutenção, e o *software*, por sua vez, se deteriora ou fica defasado, exigindo sua evolução.

AUTOATIVIDADE



1 À medida que o *hardware* evolui, o *software* também muda a fim de se tornar acessível ao usuário final. A seguir constam os principais recursos/equipamentos históricos da era do computador:

- I – Mark I.
- II – ENIAC (Computador Integrador Numérico Eletrônico).
- III – EDVAC (Computador Eletrônico com Discreta Variação Automática).
- IV – EDSAC (Calculadora Automática com Armazenamento por Retardo Eletrônico).
- V – LEO (Escritório Eletrônico de Lyons).

- () Primeiro computador, feito para fins militares para auxiliar nos cálculos de precisão necessários para balística.
- () O primeiro computador eletrônico à válvula, sua memória era muito pequena, e a cada nova operação era necessário reconfigurar toda a sua fiação, exigindo um enorme esforço humano.
- () Foi o primeiro computador eletrônico com programa armazenado utilizando sistema binário, tornando-se a arquitetura padrão para os computadores mais modernos.
- () O primeiro computador operacional em grande escala capaz de armazenar seus próprios programas.
- () O primeiro computador comercial do mundo que começou a automatizar os trabalhos de escritórios.

De acordo com as sentenças acima, assinale a resposta certa dada pela associação histórica de cada tipo de computador com sua finalidade:

- a) () I – II – III – IV – V.
- b) () I – III – II – V – IV.
- c) () II – I – IV – III – V.
- d) () I – IV – III – II – V.

2 Sobre o *software* é CORRETO afirmar:

- a) () O *software* é criado por meio de uma série de regras em linguagem de programação que especifica a estrutura de seus dados, os atributos procedimentais e os requisitos relacionados para código e convertendo em instruções executáveis em máquina.
- b) () Para a criação do *software*, os programadores utilizam uma linguagem de programação que é interpretada por um compilador que a transforma para código binário, o qual é lido pelo *hardware*.
- c) () O *software* é um conjunto de algoritmos codificados que permite ao computador executar uma operação ou um conjunto de operações culminando em tarefas.
- d) () Todas as anteriores.

3 A evolução do *software* perpassou por diversas aplicações, as quais iremos recordar a seguir.

- I – Os primeiros anos (1950 até 1960).
- II – A segunda era (1960 até 1970).
- III – A terceira era (1970 até hoje).
- IV – A quarta era (apenas começando)

- () O *hardware* dedicava-se à execução de um único programa que, por sua vez, dedicava-se a uma única aplicação específica e usava uma orientação *batch* (em lote) para a maioria dos sistemas.
- () Sistemas multiusuários ofereceram sofisticação de *software* e *hardware*, melhorando a interação homem-máquina, onde foi o período da multiprogramação.
- () Sistemas distribuídos e múltiplos computadores, onde cada um, executando funções concorrentemente e comunicando-se um com o outro, aumentou intensamente a complexidade dos sistemas baseados em computador.
- () Tecnologias orientadas a objetos, orientadas a documentos nos quais os sistemas especialistas e a inteligência artificial tornam-se prática em problemas de amplo espectro do mundo real.

De acordo com cada época e suas aplicações descritas acima, assinale a alternativa CORRETA:

- a) () I – II – III – IV.
- b) () I – II – IV – III.
- c) () II – I – III – IV.
- d) () I – III – II – IV.

4 O *software* é classificado por sete diferentes categorias. Assinale cada sentença a seguir com V para verdadeiro ou F para falso para as informações apresentadas para estas categorias.

- () O *software* básico é uma coleção de programas que dão apoio a outros programas. Exemplo: compiladores, editores simples, *drivers*, componentes do SO etc.
- () O *software* de tempo real irá monitorar, analisar e controlar eventos do mundo real.
- () O *software* comercial facilita as operações comerciais e decisões administrativas. Exemplos: controle de estoque, finanças, vendas etc.
- () O *software* científico e de engenharia trata dos algoritmos com intenso processamento de números e cálculos. Exemplos: sistemas de astronomia, naves espaciais, matemática avançada etc.
- () O *software* embutido é usado para controlar produtos e sistemas para mercados industriais e de consumo. Exemplo: controle de teclado para fornos de micro-ondas, funções digitais em automóveis etc.

- () O *software* de computador pessoal é utilizado em computadores de uso pessoal. Exemplos: editores de texto, planilhas, calculadora, jogos etc.
 - () O *software* de inteligência artificial faz uso de algoritmos não numéricos para resolver problemas complexos, que não sejam favoráveis à computação ou à análise direta. Exemplos: sistema de reconhecimento de imagem, sistemas especialistas, redes neurais e aprendizado etc.
- 5 Explique por que, com o passar do tempo, o *hardware* vai se desgastando, exigindo manutenção, e o *software* vai se deteriorando, exigindo evolução ou novo desenvolvimento.

FUNDAMENTOS DE ENGENHARIA DE SOFTWARE

1 INTRODUÇÃO

A Engenharia de *Software* originou-se conceitualmente em 1969, por Fritz Bauer, durante uma conferência patrocinada pelo Comitê de Ciência da Organização do Tratado do Atlântico Norte (OTAN), período da segunda era da evolução do *software* e momento em que a crise do *software* precisava de uma solução para que em seu desenvolvimento os projetos de *software* fossem entregues dentro de custo e prazo adequados.

Segundo Hirama (2011), nessa época a crise foi identificada pela preocupação crescente na comunidade de *software* com a quantidade de defeitos, entregas fora de prazo e altos custos do *software*. Os códigos eram difíceis de manter pela inexistência de métodos eficazes para seu desenvolvimento. Não existia solução eficaz que evitasse tantos “furos” nos projetos. A falta de qualidade do *software* não era evitada pelos desenvolvedores, porque ainda não se usava controle de qualidade no ambiente de desenvolvimento, sendo difíceis, também, de se manter e evoluir.

O termo “crise do *software*” foi usado pela primeira vez com impacto por Dijkstra (1972), o qual avaliava que, considerando o rápido progresso do *hardware* e das demandas por sistemas cada vez mais complexos, os desenvolvedores simplesmente estavam se perdendo, porque a Engenharia de *Software*, na época, era uma disciplina incipiente.

Como mencionado nesta mesma obra de Hirama (2007, p. 7), Fritz Bauer teria dito: “A Engenharia de *Software* é o estabelecimento e uso de sólidos princípios de engenharia a fim de obter um *software* que seja confiável e que funcione de forma econômica e eficiente em máquinas reais”.

Em seu livro *Engenharia de Software: Qualidade e Produtividade com Tecnologia*, Hirama (2011) explica que a existência da Engenharia de *Software* se faz importante pelos seguintes motivos:

- **A complexidade dos softwares:** especificar sistemas é uma atividade bastante complexa. Não se trata apenas de fazer uns “programinhas”.
- **Insatisfação dos usuários:** o usuário precisa de sistemas funcionando de acordo com suas necessidades e que sejam fáceis de serem operados e/ou cujo desenvolvimento não seja demorado.

- **Produtividade:** costuma estar quase sempre abaixo do desejado. Frequentemente, a alocação de recursos e atividades é desbalanceada. Custos, tempo e recursos geralmente são subestimados.
- **Confiabilidade do Sistema:** há diversas estatísticas que provam a pouca confiabilidade de boa parte dos sistemas. Não basta que o sistema produza resultados solicitados pelo usuário, mas que também tenha o desempenho adequado.
- **Manutenibilidade:** facilidade de se modificar um sistema para adaptar-se a circunstâncias novas, inexistentes à época da implantação. Sistemas recentemente implantados são substituídos por novos devido ao alto custo para sua manutenção.

Com minha experiência desde 2007 como gestor do processo de Engenharia de *Software*, posso reforçar que a Engenharia de *Software* é uma aliada indispensável às empresas de *software*, é o que define métodos sistemáticos para o desenvolvimento de *software*, buscando melhorar e amadurecer as técnicas e ferramentas utilizadas no ambiente de desenvolvimento para aumentar sua produtividade e qualidade de desenvolvimento.

Não é fácil desenvolver um *software* de qualidade, por isso é preciso criar uma disciplina aplicada a toda a equipe envolvida, começando pelos gerentes de tecnologia, diretores, analistas, programadores e a equipe de suporte e usuários do sistema. Empresas que desenvolvem *software* de qualidade são mais competitivas e podem, em geral, oferecer um melhor serviço ao seu cliente final.



Como a engenharia de *software* é uma ciência que estuda metodologias e padrões de desenvolvimento de *software*, veremos com mais detalhes na página 23 sobre a importância e os conceitos de Metodologia de Desenvolvimento de Sistemas.

Segundo Ian Sommerville (2011), a Engenharia de *Software* é uma disciplina da engenharia de sistemas que se ocupa de todos os aspectos da produção de *software*, desde os estágios iniciais de levantamento e especificação de requisitos até a implantação e manutenção, ou seja, que entrou em operação. É um conjunto de atividades, parcialmente ou totalmente ordenadas, com a finalidade de obter um produto de *software* de qualidade e cumprir corretamente os contratos de desenvolvimento.

Segundo Roger Pressman (2006), a Engenharia de *Software* poderá ser mais bem entendida como uma tecnologia em camadas ou níveis, conforme pode ser vista na Figura 10.

FIGURA 10 - CAMADAS DA ENGENHARIA DE SOFTWARE



FONTE: Disponível em: <http://3.bp.blogspot.com/_CMoqSGzMYOg/SkrJEy3nGel/AAAAA-AAAAAAk/Ee1ZgJBwMdM/s320/Figura+2.1.Engenharia+de+Software+em+Camadas.gif>. Acesso em: 06 jul. 2015

Na base da figura, formando a camada foco na qualidade, dá-se ênfase à preocupação de qualquer disciplina de engenharia, que é qualidade. A qualidade na Engenharia de *Software* é baseada nos conceitos de gerenciamento de qualidade total (TQM – *Total Quality Management*) para a melhoria contínua dos processos. O TQM é uma abordagem de gerenciamento organizacional (princípios, métodos e técnicas) para obter sucesso em longo prazo através da satisfação dos clientes.

A camada de processo permite integrar as camadas de métodos e de ferramentas para que se possa desenvolver um *software* nos prazos acordados e de maneira adequada. Um processo permite que se planeje e se controle projeto de *software*.

A camada de métodos provê as abordagens e as atividades necessárias para a construção de um *software*. Os métodos abrangem um conjunto amplo de tarefas que incluem análise de requisitos, projeto, implementação, testes e manutenção. Os métodos de Engenharia de *Software* são baseados em um conjunto de princípios que governam cada área de tecnologia e incluem atividades de modelagem e técnicas descritivas.

A camada de ferramentas provê apoio automatizado ou semiautomatizado para processos e métodos. As ferramentas da área de Engenharia de *Software* são conhecidas como CASE (Engenharia de *Software* Apoiada por Computador, do termo em inglês *Computer-Aided Software Engineering*).

FONTE: Hirama (2011, p. 8)

2 PRINCÍPIOS DA ENGENHARIA DE SOFTWARE

A aplicação da Engenharia de *Software* segue uma abordagem bem completa, onde são recomendados caminhos como referência para sua correta e benéfica utilização. Em diversos livros são apresentados como introdução à Engenharia de *Software* alguns princípios gerais aplicados durante toda a fase de desenvolvimento de *software*, da importância de algumas propriedades gerais dos processos e produtos.

A seguir são mencionados, de forma resumida, os conceitos de cada um dos 12 princípios da Engenharia de *Software* descritos por Carvalho (2001) em seu livro *Introdução à engenharia de software*:

1. **Formalidade:** deve ser desenvolvido com passos definidos e com precisão, seguidos de maneira efetiva. Não se deve restringir a criatividade, mas melhorá-la, uma vez que são criticamente analisados à luz de uma avaliação formal. Seus efeitos benéficos podem ser sentidos na manutenção, reutilização, portabilidade e entendimento do *software*.
2. **Abstração:** é o processo de identificação dos aspectos importantes de um determinado fenômeno, ignorando-se os detalhes. Os programas, por si só, são abstrações das funcionalidades do sistema.
3. **Decomposição:** uma das maneiras de lidar com a complexidade é subdividir o processo em atividades específicas, diminuindo a complexidade do problema, provavelmente atribuídas a especialistas de diferentes áreas. A decomposição das atividades leva, também, à separação das preocupações ou responsabilidades.
4. **Generalização:** pensar na resolução de uma forma generalizada para permitir reutilização. É o processo de identificação dos aspectos importantes de um determinado fenômeno, ignorando-se os detalhes.
5. **Flexibilidade:** diz respeito tanto ao processo como ao produto do *software*. O produto sofre constantes mudanças, pois, em muitos casos, a aplicação é desenvolvida incrementalmente enquanto seus requisitos ainda não foram totalmente entendidos. Devem permitir ao processo de desenvolvimento que o produto possa ser modificado com facilidade. Permitir que partes ou componentes de um produto desenvolvido possam ser utilizados em outros sistemas, bem como a sua portabilidade para diferentes sistemas computacionais.
6. **Padronização:** padronizar o processo de construção do *software* para facilitar o entendimento e manutenção.
7. **Rastreabilidade:** modo de saber o que já foi feito e o que ainda não foi feito.
8. **Desenvolvimento iterativo:** toda a equipe é engajada na solução. Exemplo: SCRUM.
9. **Gerenciamento de requisitos:** deixar claro o que deve ser atendido pelo sistema formalmente em requisitos.
10. **Arquiteturas baseadas em componentes:** separar a solução em componentes bem definidos funcionais e lógicos, com interfaces bem definidas que não compartilham estados e se comunicam por troca de mensagens contendo dados.

11. **Modelagem visual:** diagramas de fácil visualização da solução de um determinado problema. Utilizado para rápida compreensão da solução ou de um processo.
12. **Verificação contínua de qualidade:** criação de testes automatizados de modo que garantam a cobertura do código e testes de qualidade que garantam a qualidade do *software*.



Nota-se, de forma importante, que a engenharia de *software* segue os mesmos princípios de uma disciplina de engenharia tradicional, baseada em uma relação adequada de custo/benefício do produto, que não falhe e que seja eficiente, pois todos estes conceitos são necessários para atender as reais necessidades, atualmente, dos muitos exigentes usuários.

Portanto, como a tecnologia não para de crescer, é preciso buscar implementar metodologias ao ambiente de desenvolvimento de *software* nas empresas que definam atividades e técnicas a serem utilizadas em sua criação, oferecendo modelos, padrões, arquiteturas, métodos e processos que possam oferecer maior produtividade, eficiência, profissionalismo e criatividade das equipes envolvidas. Enfim, não deve se preocupar em apenas criar fórmulas e regras para chegar ao objetivo final do projeto, mas em realizar isso da melhor forma e com resultados positivos, conforme planejamento e execução destas metodologias.

Com o avanço das tecnologias e o aumento da complexidade dos *softwares*, as exigências foram aumentando e, consequentemente, as ferramentas de desenvolvimento foram surgindo e melhorando em relação à estruturação na criação dos sistemas através de suas metodologias de desenvolvimento, que serão abordadas no tópico seguinte.

3 METODOLOGIA DE DESENVOLVIMENTO DE SISTEMAS

No início da era da computação não existiam métodos para o controle e a produção do *software*, pois eram mais simples de serem feitos e não eram produzidos em grande escala. Também não existiam equipes para realizar um planejamento do desenvolvimento de *software*, o que influenciava diretamente em constantes desvios no prazo de entrega e no alto custo de produção do *software*.

Segundo Fernandes (1999), metodologia de sistemas se define como um conjunto de normas, procedimentos, técnicas e ferramentas de análise que definem o padrão desejado por uma empresa para o desenvolvimento de projetos de sistemas.

A ausência de uma metodologia de desenvolvimento de sistemas pode levar ao caos, na medida em que cada indivíduo procura aplicar em seu projeto as melhores soluções dentro das limitações de sua experiência profissional. Mesmo que suas soluções sejam ótimas e que os resultados individuais sejam melhores, dificilmente, no conjunto de todas as aplicações de uma corporação, haverá a harmonia desejada. A produtividade e a eficiência que são esperadas de um departamento de sistemas não podem ser obtidas sem critérios, sem regras e sem análise contínua das ferramentas de trabalho postas à disposição dos profissionais de sistemas.

FONTE: Fernandes (1999)

A Figura 11 apresenta que estes métodos estabelecem produtos de trabalho padronizados que facilitam as atividades de manutenção de *software*, permitindo que os colaboradores sejam treinados, melhorando a qualidade dos serviços através de um canal de comunicação uniforme entre os membros da equipe de desenvolvimento, para um maior aproveitamento dos seus recursos.

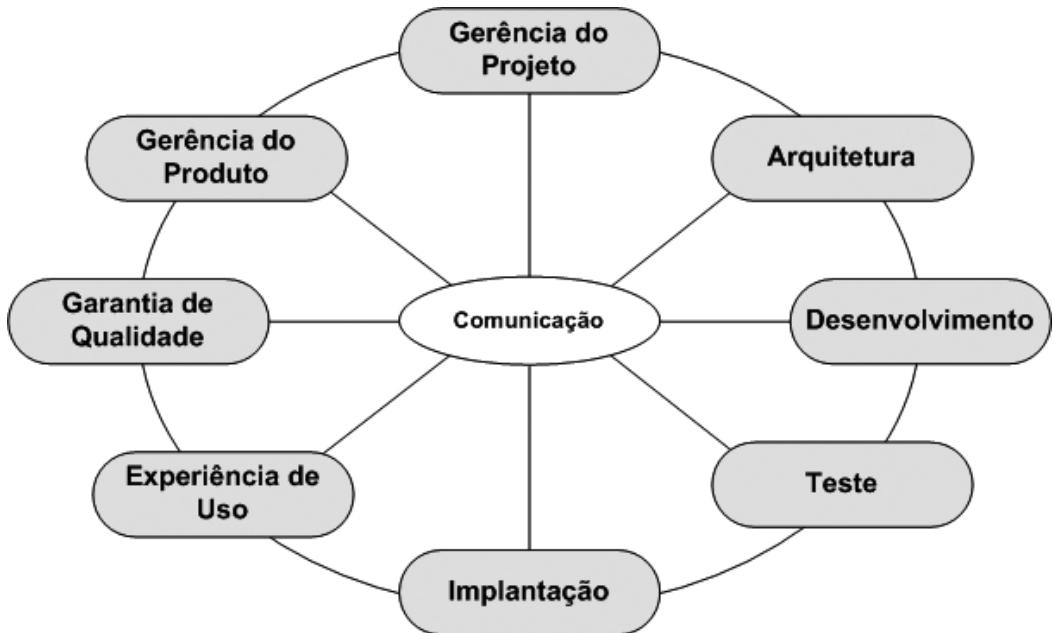
FIGURA 11 - O QUE É UMA METODOLOGIA



FONTE: Disponível em: < <http://slideplayer.com.br/slide/386223/> >. Acesso em: 24 jul. 2015

Em um projeto de *software* existem, em geral, muitos profissionais envolvidos. Têm-se, entre outros, gerentes, analistas, arquitetos, programadores e testadores. Durante a realização das atividades de desenvolvimento, a comunicação entre eles é fundamental. Para estabelecer um canal de comunicação uniforme, é necessário aplicar métodos definidos em processos de desenvolvimento de *software*.

FIGURA 12 - PROCESSO DE COMUNICAÇÃO



FONTE: O autor



As ferramentas CASE (*Computer Aided Software Engineering* ou Engenharia de Software apoiado por Computador) são programas que auxiliam atividades de engenharia de software na construção de sistemas. Desde a análise de requisitos e modelagem até programação e testes. Para identificar a "Seleção de Ferramentas CASE" recomenda-se a leitura da monografia de Elison Roberto Imenes, disponível em: <<http://bibdig.poliseducacional.com.br/document/?view=106>>.

Fernandes (1999) nos diz que, para que uma metodologia de desenvolvimento de sistemas seja consistente, oferecendo maior produtividade e qualidade, deverá atender a alguns requisitos fundamentais:

- **Padronização:** executar as atividades de maneira idêntica, fazendo com que haja aperfeiçoamento do processo.
- **Flexibilidade:** é a capacidade de se adaptar às mudanças.
- **Documentação:** manter informações sobre o produto e garantir rapidez diante das mudanças.
- **Modularização:** consiste em dividir um conjunto de atividades em vários conjuntos menores, objetivando melhor visualização e acompanhamento por parte de todos os interessados no resultado final.
- **Planejamento:** forma madura de administrar o tempo, é programar o futuro em relação às metas e aos objetivos a serem alcançados.

Gomede (2010) menciona a existência da divisão da Engenharia de Software em dez áreas de conhecimento, segundo o SWEBOK, as quais são descritas a seguir:

- **Design de Software:** Área que envolve definição da arquitetura, componentes, interfaces e outras características de um componente ou sistema. Analisando como um processo, esta é uma etapa do ciclo de vida da ES, onde é processada a análise dos requisitos com o objetivo de produzir uma descrição da arquitetura do *software*, ou seja, de forma interativa, os requisitos são traduzidos em um documento para construção do *software*.
- **Construção de Software:** Trata-se da implementação ou codificação do *software*, verificação, testes de unidade, testes de integração e depuração. Está envolvida com todas as áreas de conhecimento, porém, ligada fortemente às áreas de *Design* e *Teste de Software*, pois este processo de construção abrange significativamente ambas as áreas.
- **Teste de Software:** É uma atividade com o intuito de avaliar a qualidade do produto, buscando identificar problemas e defeitos existentes. Trata-se de um elemento crítico da garantia da qualidade de *software* e representa a verificação final da especificação, projeto e geração de código.
- **Manutenção de Software:** Área definida como o conjunto das atividades requeridas para fornecer apoio a um sistema de *software*, que pode ocorrer antes ou depois da entrega. São realizadas atividades de planejamento antes da entrega do *software*; depois, são executadas modificações com o objetivo de melhorar o desempenho, corrigir falhas, ou adaptá-las a um ambiente externo. Podemos chamar de “ajuste fino”.
- **Gerenciamento de Configuração de Software:** É um processo que provê recursos para o controle da evolução, identificação e auditagem dos artefatos de *software* gerados durante o desenvolvimento do projeto, ou seja, é o controle de versões do *software*, com a finalidade de estabelecer e manter a integridade dos produtos de *software* durante todo seu ciclo de vida.
- **Gerenciamento de Engenharia de Software:** Pode-se definir como a aplicação das atividades de gerenciamento, garantindo que o desenvolvimento e a gerência de *software* sejam sistemáticos, disciplinados e qualificados. Englobando atividades como controle, documentação, monitoramento e medição. Em suma, é a coordenação e o planejamento.
- **Engenharia de Processo de Software:** Seu objetivo é implementar processos novos e melhores, seja no escopo individual, de projeto ou organizacional, e pode ser definida como uma visão geral sobre questões do processo, amplamente relacionadas à definição, implementação, avaliação, mensuração, gerenciamento, mudanças e melhorias do processo de ciclo de vida de *software*.
- **Ferramentas e Métodos de Software:** São ferramentas criadas para prestar auxílio no ciclo de vida do *software*. Ferramentas estas que normalmente automatizam atividades do processo de desenvolvimento, auxiliando o analista no processo de concentração para as atividades que exigem

maior trabalho intelectual. Estruturam atividades de desenvolvimento e manutenção de *software* para torná-las sistemáticas e suscetíveis ao sucesso, já que seu objetivo é a pesquisa de ferramentas e métodos para o aumento da produtividade e redução de falhas.

- **Qualidade de Software:** Está diretamente ligada à qualidade do *software*. É submetido durante o processo de desenvolvimento, consequentemente, para a qualidade existir, o processo de desenvolvimento de um produto de *software* precisa ser bem definido, documentado e acompanhado. A avaliação da qualidade normalmente é feita através de modelos. Modelos estes que descrevem e alinham as propriedades de qualidade do produto. Os modelos de avaliação mais aceitos e usados no mercado são CMMI (*Capability Maturity Model Integration*), proposto pelo CMM (*Capability Maturity Model*) e a norma ISO/IEC 9126, proposta pela ISO (*International Organization for Standardization*).
- **Requisitos de Software:** Requisitos de *software* expressam a necessidade e restrições ou limitações colocadas sobre o produto ou *software* que auxiliam na solução de problemas do mundo real, expondo, analisando, especificando e validando os requisitos de *software*. Ananias (2009) também menciona que a gerência de requisitos concorre em paralelo com a engenharia de requisitos e se faz necessária para minimizar os problemas que podem ocorrer em todos os estágios do desenvolvimento de sistemas, sendo uma das mais importantes áreas da engenharia de *software*.

FONTE: Disponível em: <<http://evertongomede.blogspot.com.br/2010/08/areas-de-conhecimento-segundo-o-swebok.html>>. Acesso em: 06 jul. 2015.

3.1 FASES DE DESENVOLVIMENTO DE SOFTWARE

Atualmente, muitas são as metodologias de desenvolvimento de *softwares*. Existem as clássicas (antigas), que são mais estáveis de serem executadas através de diversos ciclos de vida prescritivo, seguindo um único caminho de trabalho, e aquelas metodologias ágeis, que possuem diversas formas dinâmicas de execução, exigindo maior experiência dos envolvidos.

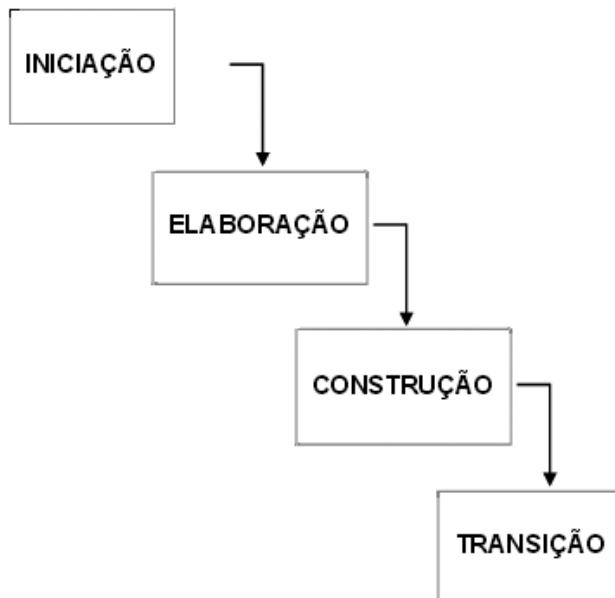
Para capturar a dimensão do tempo de um projeto, o processo de Engenharia de *Software* se divide em quatro fases que indicam a ênfase que é dada no projeto em um dado instante:

- **Fase de Iniciação:** ênfase no escopo.
- **Fase de Elaboração:** ênfase na análise.
- **Fase de Construção:** ênfase no desenvolvimento.
- **Fase de Transição:** ênfase na implantação.

Podemos colocar, ainda, que a fase de iniciação foca no tratamento de riscos relacionados com os requisitos de negócio; na fase de elaboração, o foco deve ser nos riscos técnicos e arquiteturais. O escopo deve ser revisado e os requisitos devem estar mais compreendidos do que durante a construção, a atenção será

voltada para os riscos lógicos e a maior parte do trabalho será realizada; e na fase de transição, finalmente, serão tratados os riscos associados com a logística de distribuição do produto para a base de usuários.

FIGURA 13 - FASE DE DESENVOLVIMENTO DE SOFTWARE



FONTE: O autor

Segue uma breve descrição das fases, colocando os principais itens a serem atendidos, bem como quais os principais produtos gerados em cada fase, onde em qualquer fase poderá ocorrer um replanejamento ou mudança de escopo, conforme monitoramento e controle realizados pelos coordenadores ou gerentes de projetos.

3.1.1 Fase de iniciação

- Esta fase tem como objetivo principal o planejamento do projeto e compreensão real do escopo do projeto.
- Nesta fase serão envolvidos, principalmente, os seguintes atores com suas respectivas responsabilidades:
 - Coordenador de Projeto: com a responsabilidade de elaborar e aprovar o planejamento do Projeto (o principal documento gerado é o Plano de Projeto (com seus anexos)).
 - Analista de sistema: terá a responsabilidade de refinar os Requisitos de Sistema para garantir o escopo do projeto.
 - Analista de Negócio: tem a responsabilidade de aprovar os Requisitos de Sistema que irão garantir o entendimento do escopo do projeto.

- Ao final desta fase deve-se garantir que planejamento do projeto e o escopo do projeto foram compreendidos por todos os envolvidos e aprovados por todos os responsáveis.

3.1.2 Fase de elaboração

- Esta fase tem como objetivo principal a realização da análise do projeto, onde serão criados os documentos que definirão como o projeto será implementado.
- Nesta fase será envolvido, principalmente, o seguinte ator com suas respectivas responsabilidades:
 - Analista de Sistema terá responsabilidade de definir como os requisitos de sistemas devem ser implementados, apresentando soluções para regras de negócio, banco de dados etc. O analista poderá representar a solução através do Documento de Caso de Uso e/ou documento de Especificação de Implementação, entre outros.
- Ao final desta fase deve-se garantir que a solução para análise está representada da melhor forma, que irá possibilitar ao Programador realizar seu trabalho, bem como ao Analista de Teste.

3.1.3 Fase de construção

- Esta fase tem como objetivo principal a materialização da análise, pois será realizada a implementação dos componentes que irão compor o projeto. Nesta fase, também, serão realizados os testes.
- Nesta fase será envolvido, principalmente, o seguinte ator com suas respectivas responsabilidades:
 - Implementador: terá responsabilidade de implementar os componentes e realizar os testes de unidade. O principal produto gerado será o código-fonte.
- Ao final desta fase deve-se garantir que os requisitos solicitados foram implementados conforme a solicitação.

3.1.4 Fase de transição

- Esta fase tem como objetivo principal realizar a entrega do projeto.
- Nesta fase será envolvido, principalmente, o seguinte ator com suas respectivas responsabilidades:
 - Analista de Negócio: terá responsabilidade de aprovar a entrega do produto realizando os testes de aceite.
- Ao final desta fase deve-se garantir a entrega do produto atendendo ao escopo e ao planejamento.

4 PROCESSOS DE ENGENHARIA DE SOFTWARE

Quando se fala em metodologia de desenvolvimento, deve-se pensar em institucionalizá-lo através de um processo de *software*. Um produto com qualidade só é possível se for construído de forma bem organizada, seguindo uma série de passos previsíveis através de um guia que ajude a controlar e chegar a um resultado de qualidade no prazo previsto.

A implantação de um processo pode ser visto como o conjunto de atividades, métodos, ferramentas, práticas e transformações que guiam pessoas na produção de *software* de forma mais assertiva e desenvolvimento ágil.

Um modelo de ciclo de vida organiza as macroatividades básicas, estabelecendo precedência e dependência entre elas. Um processo eficaz deve, claramente, considerar as relações entre as atividades, as pessoas que as executam (habilidades, treinamentos e motivação), os artefatos produzidos no desenvolvimento, as ferramentas e os procedimentos necessários.

No momento em que se decide construir um *software*, é fundamental também decidir qual processo será seguido. A criação de um projeto de *software* é uma atividade intelectual de alto nível de complexidade, necessitando melhor visibilidade na sua construção, sabendo-se de início quais são as etapas do projeto.

Aparentemente, todo e qualquer projeto de *software* pode ser construído sem que seja utilizado formalmente um guia predefinido para executá-lo. Porém, isso só é possível quando o projeto for construído como um passatempo ou *hobby* por uma ou duas pessoas em poucas horas de trabalho. Já no desenvolvimento profissional de *software*, o cenário pode ser radicalmente diferente e limitado, requeira muito retrabalho, tornando-o inviável caso não haja a utilização de um processo de *software* que reduza, dentro do possível, o desperdício de tempo e maximize os resultados, obtendo alta qualidade e satisfação em construí-lo.

Um processo de *software* é composto por métodos (aquilo que diz o que, em uma determinada tarefa), por ferramentas (que dão suporte automatizado aos métodos) e procedimentos (que fazem o elo de ligação entre os métodos e as ferramentas). Uma organização que possui um processo de engenharia de *software* deverá levar muito a sério estes três princípios de processo, a fim de que seus projetos de *software* sejam de sucesso, ou seja, equipes produtivas e *softwares* bem feitos.

Para facilitar as atividades dos analistas de processo nas organizações, geralmente o processo de Engenharia de *Software* é decomposto em diversos processos, tais como processo de gerência de projeto, processo de modelagem, processo de implementação, processo de qualidade, processo de gerência de configuração etc., que estão interligados e servem de apoio um ao outro durante o ciclo de vida do desenvolvimento do *software*.

Verifique, através da figura a seguir, que cada processo, por sua vez, é composto de atividades que também podem ser decompostas em pré-atividades,

subatividades, compostas por artefatos de entrada (insumos) e de saída (produtos) da atividade, os recursos necessários (humanos, *hardware*, *software* etc.) e os procedimentos (métodos, técnicas, modelos de documento etc.) a serem utilizados na sua realização.

FIGURA 14 – ELEMENTOS QUE COMPÕEM UM PROCESSO DE SOFTWARE

Processo de Software	Exemplo:
Processos	
Atividades	
Pré-atividades	Atv1. Realizar levantamento de requisitos.
Subatividades	<i>Pré-Atividade: Atv0</i> <i>Insumo: Documento Planejamento de Entrevista</i> <i>Produto: Documento Registro de Entrevista.</i>
Artefatos	<i>Recurso Humano: Analista de Sistemas</i> <i>Procedimento: Técnica para Realização de Entrevistas</i>
Insumentos	
Produtos	
Recursos	
Recursos Humanos	Atv2. Documentar requisitos.
Ferramentas de Software	<i>Pré-Atividade: Atv1</i> <i>Insumo: Documento Registro de Entrevista.</i> <i>Produto: Especificação Textual de Requisitos.</i>
Hardware	<i>Recurso Humano: Analista de Sistemas</i> <i>Procedimento: Roteiro para Elaboração da Especificação Textual de Requisitos.</i>
Procedimentos	
Métodos	
Técnicas	
Roteiros	

FONTE: Disponível em: <<http://www.inf.ufes.br/~monalessa/PaginaMonalessa-NEMO/ES/Slides-1-ES-Introducao&ProcessoSoftware.pdf>>. Acesso em: 06 jul. 2015.

Para elaboração de um processo de *software* é importante que a organização possua uma boa ferramenta de documentação, que tenha inteligibilidade durante suas definições, permitindo que o progresso do processo seja visível externamente, tenha suporte a outras ferramentas CASE, com boa aceitabilidade e confiabilidade durante a utilização. Seja robusto para continuá-lo a despeito de problemas inesperados e tenha boa facilidade de manutenção e velocidade para construí-lo.

FIGURA 15 - ELABORAÇÃO DE PROCESSO DE SOFTWARE



FONTE: Disponível em: <http://www.projectbuilder.com.br/blog-pb/entry/dicas/5-motivos-para-documentar-seus-processos-e-adotar-uma-ferramenta-de-gp>. Acesso em: 26 jul 2015.

Para apoiar a definição de processos de *software*, diversas normas e modelos de qualidade foram propostos, dentre elas: ISO 9001, ISO/IEC 12207, ISO/IEC 15504, CMMI e MPS.Br - cujo objetivo é apontar características que um bom processo de *software* tem de apresentar, deixando a organização livre para estruturar essas características segundo sua própria cultura.



Maiores detalhes sobre as normas e modelos de qualidade serão tratados na Unidade 3 deste Caderno de Estudos.

Com o aumento da complexidade dos processos de *software*, passou a ser imprescindível o uso de ferramentas e ambientes de apoio à realização de suas atividades, visando, sobretudo, atingir níveis mais altos de qualidade e produtividade. Ferramentas CASE (*Computer Aided Software Engineering*) passaram, então, a ser utilizadas para apoiar a realização de atividades específicas, tais como planejamento e análise e especificação de requisitos.



Diversas ferramentas CASE (*Computer Aided Software Engineering*) podem ser encontradas no mercado via framework, que trata sobre sua aplicação a fim de auxiliar no desenvolvimento de *software*. Durante este caderno de estudos você irá perceber alguns frameworks nas áreas de gerência de configuração, planejamento, codificação e testes de *software*.

Apesar dos benefícios do uso de ferramentas CASE individuais, atualmente, o número e a variedade de ferramentas têm crescido a tal ponto que levou os ES a pensarem não apenas em automatizar os seus processos, mas sim em trabalhar com diversas ferramentas que interajam entre si e forneçam suporte a todo ciclo de vida do desenvolvimento, que buscam combinar técnicas, métodos e ferramentas para apoiar o engenheiro de *software* na construção de produtos de *software*, abrangendo todas as atividades inerentes ao processo: gerência, desenvolvimento e controle da qualidade.

A escolha de um modelo de processo (ou modelo de ciclo de vida) é o ponto de partida para a definição de um processo de desenvolvimento de *software*. Um modelo de ciclo de vida organiza as macroatividades básicas, estabelecendo precedência e dependência entre elas. Maiores detalhes sobre modelos de ciclo de vida de *software* serão apresentados no próximo tópico desta Unidade 1.

LEITURA COMPLEMENTAR

CÓDIGO DE ÉTICA UNIFICADO IEEE-CS/ACM PARA ENGENHARIA DE SOFTWARE

Este código foi desenvolvido pela força-tarefa conjunta do IEEE-CS/ACM e foi publicada em Communications of the ACM, vol. 42:10, Oct. 1999. Sua versão *on-line* está disponível em: <<http://www.computer.org/labseprof/cod.htm>>.

Devido a seus papéis no desenvolvimento de *software*, os Engenheiros de *Software* têm oportunidades significativas para fazer o bem ou para causar o mal, para permitir que outros façam o bem ou causem o mal, ou para influenciar outros a fazer o bem ou causar o mal. Para assegurar, tanto quanto possível, que seus esforços serão usados para o bem, os engenheiros de *software* devem comprometer-se a fazer da engenharia de *software* uma profissão benéfica e respeitada.

O código contém **OITO PRINCÍPIOS** relacionados ao comportamento e decisões tomadas pelos Engenheiros de *Software*, incluindo praticantes, educadores, gerentes, supervisores e criadores de políticas, assim como aprendizes e estudantes da profissão. Os princípios identificam as relações em que indivíduos, grupos e organizações participam e as principais obrigações pertinentes a essas relações. As cláusulas desses princípios são ilustrações de algumas das obrigações incluídas nesses relacionamentos. Essas obrigações fundamentam-se na humanidade do Engenheiro de *Software*, nos cuidados especiais devido às pessoas afetadas pelo trabalho do Engenheiro de *Software* e nos elementos únicos da prática de engenharia de *software*. O código prescreve essas obrigações para todo Engenheiro de *Software* ou qualquer um que aspire a sê-lo.

Segue relação completa dos oitos princípios:

1. Princípio 1º: PÚBLICO

Engenheiros de *software* devem atuar consistentemente com os interesses públicos. Em particular, engenheiros de *software* devem, apropriadamente:

1. Aceitar total responsabilidade pelo seu próprio trabalho.
2. Moderar os interesses do engenheiro de *software*, do empregado, do cliente, e dos usuários, com o bem público.
3. Aprovar *software* somente se o mesmo estiver absolutamente convicto que seja seguro, de acordo com suas especificações, passe nos testes apropriados, e não diminua a qualidade de vida, diminua a privacidade, ou prejudique o meio ambiente. O efeito final do trabalho deve ser pelo bem público.
4. Esclarecer para as pessoas ou autoridades apropriadas qualquer perigo real ou potencial ao usuário, ao público, ou ao meio ambiente, que se creia esteja associado ao *software* ou documentos relacionados.
5. Cooperar com os esforços para enviar desculpas a prejuízos graves causados ao público por *software*, sua instalação, manutenção, suporte, ou documentação.

6. Ser justo e evitar obter vantagens em todos os assuntos, particularmente os públicos, referentes a *software* ou documentos relacionados, métodos e ferramentas.
7. Considere questões de falta de habilidade física, alocação de recursos, desvantagens econômicas e outros fatores que podem diminuir ao acesso aos benefícios do *software*.
8. Seja encorajado a participar voluntariamente de tarefas por boas causas e para contribuir com a educação pública a favor da disciplina.

2. Princípio 2º: Cliente e Empregado

Engenheiros de *software* devem atuar de um modo que atenda os melhores interesses de seu cliente e empregado, consistentemente com os interesses públicos. Em particular, engenheiros de *software* devem, apropriadamente:

1. Prover trabalho nas suas áreas de competência, sendo honestos e transparentes sobre qualquer limitação sobre sua experiência e educação.
2. Não utilizar *software* conscientemente que tenha sido obtido ou retido de modo ilegal ou antiético.
3. Usar a propriedade do cliente ou do empregado somente nos modos em que esteja autorizado, e com o conhecimento e consentimento do empregado.
4. Estar certo de que qualquer documento sobre o qual se dependa esteja aprovado, quando necessário, por alguém autorizado pela sua aprovação.
5. Manter em segredo qualquer informação confidencial obtida no seu trabalho profissional, contanto que tal confidencialidade esteja consistente com os interesses públicos e consistentes com a lei.
6. Identificar, documentar, coletar evidências, e reportar ao cliente ou ao empregador, prontamente se, na opinião dos mesmos, um projeto esteja a ponto de falhar, de se provar muito caro, violar a lei de propriedade intelectual, ou problemático em qualquer outro ponto de vista.
7. Identificar, documentar, e reportar assuntos significativos de interesse social, dos quais esteja alertado, em *software* ou documentos relacionados, ao empregador ou ao cliente.
8. Não aceitar trabalho externo em detrimento ao trabalho executado ao seu empregador primário.
9. Não promover interesses adversos ao seu empregador ou ao seu cliente, a não ser que um interesse de valor ético maior esteja sendo compromissado; neste caso, informe ao empregador ou a outra autoridade apropriada sobre o envolvimento ético.

3. Princípio 3º: Produto

Engenheiros de *software* devem assegurar que seus produtos e modificações relacionadas estejam de acordo com os maiores padrões profissionais possíveis. Em particular, engenheiros de *software* devem, apropriadamente:

1. Comprometer com a alta qualidade, custos aceitáveis, e uma sequência de atendimento racional, assegurando que os acordos entre as partes estejam claros e sejam aceitos pelo empregado e pelo cliente, e estejam disponíveis para avaliação do usuário e do público.

2. Assegurar metas e objetivos apropriados, para cada projeto em que se trabalha ou se efetue proposta.
3. Identificar, definir e objetivar assuntos éticos, econômicos, culturais, legais e de meio ambiente, relativos a projetos de trabalho.
4. Assegurar que estejam qualificados para qualquer projeto em que trabalhem ou se proponham trabalhar, por uma combinação apropriada de educação, treinamento e experiência.
5. Assegurar que um método apropriado é utilizado para qualquer projeto no qual trabalhem ou se proponham trabalhar.
6. Trabalhar para acompanhar padrões profissionais, quando disponíveis, que sejam mais apropriados para as tarefas à mão, afastando-se deles somente quando houver justificativa ética ou técnica.
7. Comprometer-se para entender completamente as especificações do *software* no qual estão trabalhando.
8. Assegurar que as especificações para o *software* no qual esteja trabalhando tenha sido bem documentado, satisfazendo os requerimentos dos usuários e tenham as aprovações apropriadas.
9. Assegurar estimativas quantitativas realistas de custos, atendimento, pessoas, qualidade, e saídas de cada projeto em que se trabalha ou propõem trabalhar e prover transparência ao acesso a estas estimativas.
10. Assegurar adequados testes, depuração e revisões do *software* e documentos relacionados em que se trabalha.
11. Assegurar documentação adequada, incluindo problemas significativos descobertos e soluções adotadas, para qualquer projeto em que se trabalha.
12. Trabalhar para desenvolver *software* e documentos relacionados que respeitem a privacidade de todos que possam ser afetados pelo *software*.
13. Ser cuidadoso para usar somente dados precisos derivados por significados éticos e legais, e usá-los somente de modo apropriadamente autorizado.
14. Manter a integridade dos dados, sendo sensível a dados com ocorrências vencidas ou erradas.
15. Tratar todas as formas de manutenção de *software* com o mesmo profissionalismo de um novo desenvolvimento.

4. Princípio 4º: Julgamento

Engenheiros de *software* devem manter integridade e independência nos seus julgamentos profissionais. Em particular, engenheiros de *software* devem, apropriadamente:

1. Ponderar todos os julgamentos técnicos pela necessidade de suportar e manter valores humanos.
2. Somente endossar documentos preparados sob sua supervisão ou por suas áreas de competência e com os quais estejam de acordo.
3. Manter objetividade profissional com respeito a qualquer *software* ou documento relacionado que seja chamado a avaliar.
4. Não se envolver em práticas financeiras enganadoras como subornos, duplo faturamento ou outras práticas financeiras impróprias.
5. Divulgar para todas as partes envolvidas aqueles conflitos de interesses dos quais não se possa racionalmente sair ou que não possam ser evitados.

6. Recusar participar, como um membro ou orientador, em grupo profissional, privado ou governamental em assuntos relativos a *software* em que seus empregados, ou seus clientes tenham conflitos de interesses em potencial.

5. Princípio 5º: Administração

Administradores e líderes de engenheiros de *software* devem creditar e promover uma postura ética no gerenciamento do desenvolvimento e manutenção de *software*. Em particular, estes gerentes ou líderes de engenheiros de *software* devem, apropriadamente:

1. Assegurar o bom gerenciamento para qualquer projeto no qual trabalhe, incluindo procedimentos efetivos para promover a qualidade e reduzir o risco.
2. Assegurar que engenheiros de *software* sejam informados dos padrões antes de serem responsabilizados pelos mesmos.
3. Assegurar que engenheiros de *software* conheçam as políticas de emprego e procedimentos para proteção de senhas, arquivos e informações que são confidenciais ao empregado ou confidenciais a outros.
4. Determinar trabalho somente depois de ter levado em conta apropriadas contribuições de educação e experiência, equilibradas com o desejo de promover mais educação e experiência.
5. Assegurar estimativas quantitativas realistas de custo, tempo de atendimento, pessoas, qualidade e saídas em qualquer projeto no qual se trabalhe ou se proponha trabalhar, e prover transparência ao acesso a estas estimativas.
6. Atrair engenheiros de *software* potenciais somente pela completa e precisa descrição das condições do emprego.
7. Oferecer remuneração justa e racional.
8. Não prejudicar injustamente alguém de obter uma posição para a qual a pessoa esteja perfeitamente qualificada.
9. Assegurar que haja concordância racional sobre a criação de qualquer *software*, processo, pesquisa, escrita, ou outra propriedade intelectual para a qual um engenheiro de *software* tenha contribuído.
10. Prover condições para ouvir acusações de violação de uma política ou deste código pelo empregado.
11. Não pedir a nenhum engenheiro de *software* para fazer nada inconsistente com este Código.
12. Não punir ninguém por expressar opiniões éticas sobre um projeto.

6. Princípio 6º: Profissão

Engenheiros de *software* devem desenvolver a integridade e reputação da profissão consistentemente com os interesses públicos. Em particular, engenheiros de *software* devem, apropriadamente:

1. Ajudar a desenvolver um ambiente organizacional favorável a agir eticamente.
2. Promover conhecimento público de engenharia de *software*.
3. Estender conhecimento de engenharia de *software* pela participação apropriada em organizações profissionais, encontros e publicações.

4. Suportar, como membro de uma profissão, outros engenheiros de *software* empenhados em seguir este Código.
5. Não promover seu interesse próprio em detrimento da profissão, cliente ou empregado.
6. Obedecer todas as leis que governam seu trabalho, a não ser, em circunstâncias excepcionais, que seu cumprimento seja inconsistente com o interesse público.
7. Ser preciso ao descrever as características do *software* no qual se trabalha, evitando não somente assertivas falsas, mas também assertivas que podem ser racionalmente percebidas como especulativas, vagas, enganadoras, imprecisas ou falsas.
8. Assumir responsabilidade na detecção, correção e reporte de erros em *software* e documentação associada nos quais esteja trabalhando.
9. Assegurar que clientes, empregados e supervisores conheçam a concordância dos engenheiros de *software* a este Código de Ética, e as subsequentes ramificações de tal concordância.
10. Evitar associações com negócios e organizações que estejam em conflito com este Código.
11. Reconhecer que violações deste Código são inconsistentes com a profissão de engenheiro de *software*.
12. Expressar opiniões às pessoas envolvidas quando violações significativas a este Código são detectadas, a não ser que seja impossível, contraproducente ou perigoso.
13. Reportar violações significativas deste Código a autoridades apropriadas quando está claro que a consulta a pessoas envolvidas nestas significativas violações seja impossível, contraproducente ou perigosa.

7. Princípio 7º: Coleguismo

Engenheiros de *software* devem ser justos e dar suporte aos seus colegas. Em particular, engenheiros de *software* devem, apropriadamente:

1. Encorajar colegas a aderirem a este Código.
2. Assessorar colegas no seu desenvolvimento profissional.
3. Creditar amplamente o trabalho de outros e evitar aceitar créditos não merecidos.
4. Revisar o trabalho de outros de uma forma objetiva, candida e de modo apropriadamente documentado.
5. Ouvir de modo justo, opiniões, assertivas ou reclamações de colegas.
6. Ajudar colegas em estarem totalmente a par das atuais práticas de trabalho incluindo políticas e procedimentos de proteção com senhas, arquivos e outras informações confidenciais, e medidas de segurança em geral.
7. Não interferir injustamente na carreira de nenhum empregado; entretanto, consideração pelo empregado, pelo cliente, ou pelos interesses públicos incentivarão engenheiros de *software*, em boa fé, a questionar a competência de um colega.
8. Em situações fora de suas próprias áreas de competência, buscar por opiniões de outros profissionais que tenham competência nestas áreas.

8. Princípio 8º: Indivíduo

Engenheiros de *software* devem participar de aprendizado por toda sua vida otimizando a prática da sua profissão e devem promover uma abordagem ética da profissão. Em particular, engenheiros de *software* devem continuamente se esforçar por:

1. Evoluir seus conhecimentos em desenvolvimentos de análise, especificação, projeto, desenvolvimento, manutenção e teste de *software* a documentação relacionada, junto com o gerenciamento do processo.
2. Melhorar sua habilidade para criar *software* de qualidade, seguro, confiável, e útil por custos racionais em tempo racional.
3. Melhorar sua habilidade para produzir documentação precisa, informativa e bem escrita.
4. Melhorar seu entendimento sobre *software* e documentos relativos ao trabalho em que trabalha, e do ambiente em que será usado.
5. Melhorar seu conhecimento de padrões relevantes e leis que governam o *software* e a documentação relacionada em que trabalha.
6. Melhorar seu conhecimento deste Código, sua interpretação, e sua aplicação no seu trabalho.
7. Não dar tratamento injusto a ninguém por causa de alguma acusação irrelevante.
8. Não influenciar outros a tomarem nenhuma ação que envolva uma quebra deste Código.
9. Reconhecer que as violações pessoais deste Código são inconsistentes com um profissional de engenharia de *software*.

O código de ética de engenharia de *software* está de acordo com princípios universais de ética, onde qualquer conduta aceita como padrão ético deve valer para todos os que se encontram na mesma situação, sem exceções, e que só se deve exigir dos outros o que exigimos de nós mesmos, e deve-se desejar aos outros o que de melhor espera-se para nós.

FONTE: Disponível em <<http://www.computer.org/labseprof/cod.htm>>. Acesso em: 24 jul. 2015.

RESUMO DO TÓPICO 2

Neste tópico, você aprendeu que:

- A Engenharia de *Software* surgiu para auxiliar os profissionais de *software* nas crescentes dificuldades encontradas no processo de produção de soluções de *software*. A partir dos anos 60, a crise de *software* estourou devido ao fato de cada vez mais haver maior complexidade dos *softwares*, com alto índice de manutenibilidade, elevando o custo para mantê-lo.
- A Engenharia de *Software* foi fundada em 1969 por Fritz Bauer, a partir da necessidade de formalização e aperfeiçoamento dos métodos de construção de *software* para que seus projetos fossem entregues dentro de custo e prazo adequados.
- Ian Sommerville (2011) define a Engenharia de *Software* como uma disciplina da ES que se ocupa de todos os aspectos da produção de *software*, desde o início até o fim do desenvolvimento, contida por um conjunto de atividades parcialmente ou totalmente ordenadas para obter um *software* de qualidade.
- Roger Pressman entende que ES é composta por quatro camadas ou níveis: (1) foco na qualidade para obter satisfação dos clientes; (2) processos para definir as atividades, suas sequências e procedimentos; (3) métodos para definir as técnicas de execução das tarefas; (4) ferramentas para permitir a automatização dos processos e métodos de execução do projeto.
- Um processo de *software* não pode ser definido de forma universal. Para ser eficaz e conduzir a construção de produtos de boa qualidade, um processo deve ser adequado ao domínio da aplicação e ao projeto específico. Deste modo, processos devem ser definidos caso a caso, considerando-se as especificidades da aplicação, a tecnologia a ser adotada na sua construção, a organização onde o produto será desenvolvido e o grupo de desenvolvimento.
- A Engenharia de *Software* é composta pelos princípios de formalidade, abstração, decomposição, generalização, flexibilidade, padronização, rastreabilidade, desenvolvimento iterativo, arquiteturas baseadas em componentes, modelagem visual e verificação contínua de qualidade, cujo objetivo é implementar metodologias que definam a forma de desenvolvimento de *software* através de modelos, padrões, arquiteturas, métodos e processos que possam oferecer maior produtividade, eficiência, profissionalismo e criatividade das equipes envolvidas.
- A metodologia de desenvolvimento de *software* irá estabelecer produtos de trabalho padronizados que facilitem as atividades de manutenção de *software*, permitindo que os colaboradores sejam treinados, melhorando a qualidade

dos serviços através de um canal de comunicação uniforme entre os membros da equipe de desenvolvimento, para um maior aproveitamento dos seus recursos.

- Fernandes (1999) explica que, para uma boa metodologia de desenvolvimento de sistemas, é preciso haver consistência nos seguintes requisitos durante sua construção e utilização: padronização, flexibilidade, documentação, modularização e planejamento.
- A metodologia SWEBOK nos apresenta que a Engenharia de *Software* é composta por dez disciplinas, as quais compõem as áreas de conhecimento: (1) *Design de Software*, (2) Construção de *Software*, (3) Teste de *Software*, (4) Manutenção de *Software*, (5) Gerência de Configuração de *Software*, (6) Gerenciamento de ES, (7) Engenharia de Processo de *Software*, (8) Ferramentas e Métodos de *Software*, (9) Qualidade de *Software* e (10) Requisito de *Software*.
- Existem quatro fases de desenvolvimento de *software*, sendo que na fase de iniciação a ênfase é identificar o escopo do projeto; na fase de elaboração, a ênfase é análise e planejamento; na fase de construção, a ênfase é a codificação; e na fase de transição, a ênfase é a implantação.
- A construção de forma bem organizada de projeto de *software* só é possível através de um processo de *software* que pode ser visto como um conjunto de atividades, métodos, ferramentas que guiem as equipes na produção de *software* de produtividade e qualidade. Normalmente, um processo é composto de atividades, pré-atividades, subatividades, artefatos, recursos necessários e os procedimentos a serem utilizados na sua realização.
- A criação de um *software* é uma atividade intelectual de alto nível de complexidade, que exigirá o uso de processo para maior visibilidade do seu andamento e maior assertividade durante a tomada de decisão. Um processo é composto de métodos que relatam como fazer o projeto, de ferramentas que mostrem como automatizar os métodos e de procedimentos que documentem como a ferramenta executa os métodos, a fim de disseminar estas funções como guia padrão de desenvolvimento.
- As metodologias pesadas devem ser aplicadas apenas em situações em que os requisitos do *software* são estáveis e requisitos futuros são previsíveis. Estas situações são difíceis de serem atingidas, uma vez que os requisitos para o desenvolvimento de um *software* são mutáveis.

AUTOATIVIDADE



- 1 Conceitue o que é Engenharia de *Software* e diga por que ela existe.
- 2 A _____ é o estabelecimento e uso de sólidos princípios de engenharia a fim de obter um *software* que seja confiável e que funcione de forma econômica e eficiente em máquinas reais.

Assinale a alternativa que corresponde à definição CORRETA:

- a) () Engenharia de Requisitos.
- b) () Engenharia de Projetos.
- c) () Engenharia de *Software*.
- d) () Engenharia de Processos.

- 3 Quais são os principais motivos pela existência da crise de *software* a partir da década de 60?
- 4 A engenharia de *software* é uma aliada indispensável às empresas de *software*. A respeito das características e da importância, assinale V para verdadeiro ou F para falso.

- () É quem define métodos sistemáticos para o desenvolvimento de *software*, buscando melhorar e amadurecer as técnicas e ferramentas utilizadas no ambiente de desenvolvimento para aumentar sua produtividade e qualidade de desenvolvimento.
- () É uma disciplina da engenharia de sistemas que se ocupa de todos os aspectos da produção de *software*, desde os estágios iniciais de levantamento e especificação de requisitos até a implantação e manutenção,
- () É um conjunto de atividades, parcialmente ou totalmente ordenadas, com a finalidade de obter um produto de *software* de qualidade e cumprir corretamente os contratos de desenvolvimento.
- () A Engenharia de *Software* originou-se conceitualmente no período da quarta era da evolução do *software*, fora do período em que a crise do *software* existia, momento da busca de desenvolvimento ágil para que projetos fossem entregues dentro de custo e prazo adequados.

A alternativa CORRETA é:

- a) () V – F – F – V.
- b) () V – F – V – F.
- c) () V – V – V – F.
- d) () F – V – V – F.

5 Roger Pressman definiu que a Engenharia de *Software* é composta por uma tecnologia em camadas, com foco em:

- I - Qualidade.
- II - Processo.
- III - Métodos.
- IV - Ferramentas.

- () Dá-se ênfase ao apoio automatizado ou semiautomatizado para processos e métodos.
- () Dá-se ênfase às abordagens e às atividades necessárias para a construção de um *software*.
- () Dá-se ênfase ao planejamento das atividades e ao controle do projeto de *software*.
- () Dá-se ênfase à preocupação da disciplina, padronização e satisfação dos clientes.

De acordo com as sentenças acima, associe a sequência correta das definições dadas para cada camada:

- a) () I – II – III – IV.
- b) () I – III – II – IV.
- c) () II – I – IV – III.
- d) () IV – III – II – I.

6 Exemplifique cada um dos 12 princípios da Engenharia de *Software* descritos por Carvalho (2001), em seu livro *Introdução à Engenharia de Software*: Formalidade, Abstração, Decomposição, Generalização, Flexibilidade, Padronização, Rastreabilidade, Desenvolvimento iterativo, Gerenciamento de requisitos, Arquiteturas baseadas em componentes, Modelagem visual, Verificação contínua da qualidade.

7 Defina o que é e para que serve a Metodologia de Desenvolvimento de Sistemas.

8 Explique por que a ausência de uma metodologia de desenvolvimento de sistemas pode levar ao caos, na medida em que cada indivíduo procura aplicar em seu projeto as melhores soluções dentro das limitações de sua experiência profissional.

9 A respeito das metodologias de desenvolvimento de *software*, assinale V para verdadeiro ou F para falso:

- () Durante a realização das atividades de desenvolvimento, a comunicação entre os profissionais é fundamental, devendo-se estabelecer um canal de comunicação uniforme através de um método definido via processo de desenvolvimento.

- () Ferramentas CASE (*Computer Aided Software Engineering*) auxiliam atividades de engenharia de *software* na construção de sistemas, desde a análise de requisitos e modelagem até programação e testes.
- () A avaliação da qualidade dos projetos de *softwares* normalmente é feita através das metodologias, processos e ciclos de vidas adotados nos projetos, pois descrevem e alinhram as propriedades de qualidade do produto.
- () Atualmente, muitas são as metodologias de desenvolvimento de *softwares*. Existem as clássicas (antigas), que são mais estáveis, seguindo um único caminho de trabalho, e aquelas metodologias ágeis, que possuem diversas formas dinâmicas de execução, exigindo maior experiência dos envolvidos.

A sequência correta é:

- a) () V – F – F – V.
- b) () V – F – V – F.
- c) () V – V – V – V.
- d) () F – V – V – F.

10 Quais são as dez disciplinas que envolvem a Engenharia de *Software* segundo SWEBOK?

11 Um projeto de *software* é dividido em quatro grandes fases, que definem os marcos do progresso do projeto mediante seus ciclos de vida através das fases:

- I - Iniciação.
- II - Elaboração.
- III - Construção.
- IV - Transição.

- () Deve garantir a entrega completa do produto, atendendo ao escopo e ao planejamento.
- () Definição do escopo do projeto, das equipes envolvidas e com atenção voltada para os riscos lógicos.
- () É a materialização da análise através da existência dos componentes que irão compor o projeto.
- () Deve garantir a realização da análise do projeto, onde serão criadas e documentadas as necessidades do usuário para implementação.

Assinale a associação CORRETA dos itens apresentados acima:

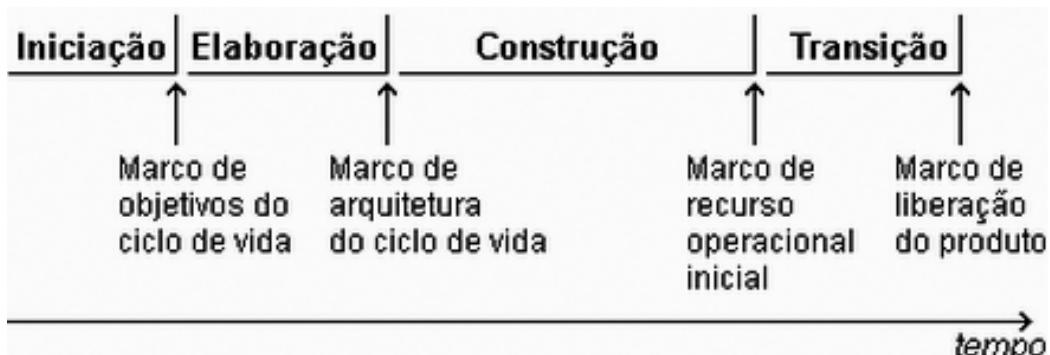
- a) () IV – II – III – I.
- b) () I – IV – III – II.
- c) () II – I – III – IV.
- d) () IV – I – III – II.

CICLO DE VIDA DE DESENVOLVIMENTO DE SOFTWARE

1 INTRODUÇÃO

Um ciclo de vida de desenvolvimento de *software* pode ser entendido como um roteiro de trabalho executado durante um projeto, no qual, em geral, cada atividade é constituída de macroetapas interdependentes, que fazem uso de métodos, técnicas, ferramentas e procedimentos para construção do produto de *software*, de forma adequada, seguindo critérios de qualidade e formando uma base sólida para o desenvolvimento.

FIGURA 16 - MACROETAPAS DO CICLO DE VIDA DE DESENVOLVIMENTO DE SOFTWARE



FONTE: Disponível em: <http://www.wthreex.com/rup/process/workflow/manageme/images/co_phas1.gif>. Acesso em: 20 jul. 2015

O ciclo de vida de um *software* descreve as fases pelas quais o *software* passa desde a sua concepção (iniciação) até ficar sem uso algum (após transição), determinando os passos a serem seguidos no desenvolvimento de sistemas, mantendo uma padronização de trabalho e determinando as etapas de validação do projeto.

De maneira geral, as fases do ciclo de vida de um *software* são constituídas de planejamento, análise e especificação de requisitos, projeto, implementação, testes, entrega e implantação, operação e manutenção.

Uma vez estabelecido o escopo do projeto, se planeja o cronograma com as estimativas de recursos, custos e prazos do projeto. Durante a análise e

especificação, o escopo é refinado, relatando o que tem que ser feito para que na fase de projeto sejam modelados os requisitos tecnológicos. Identificando-se os principais componentes arquiteturais, cada unidade de *software* é implementada e, finalmente, o sistema como um todo deve ser testado.

Após o *software* ser homologado, deve ser entregue. Para tal, é necessário treinar os usuários, configurar o ambiente de produção e, muitas vezes, converter bases de dados para o processo de operação ser iniciado, quando o *software* passa a ser utilizado pelos usuários.

Por fim, após ter sido entregue para o usuário, sempre haverá a fase de manutenção do projeto, seja por identificação de erros, pelo *software* precisar ser adaptado para acomodar mudanças em seu ambiente externo, ou pela necessidade do cliente de funcionalidade adicional ou aumento de desempenho (FALBO, 2005).

2 MODELOS DE PROCESSO DE CICLO DE VIDA DE SOFTWARE

Os modelos de processo de ciclo de vida durante o desenvolvimento do *software* podem ser linear, incremental ou iterativo, logo, compreendê-los poderá auxiliar na adoção de um dos modelos mais adequados à realidade e necessidade da organização.

No modelo linear, o *software* é executado e entregue com todas as suas funcionalidades em apenas uma fase, mais simples de utilizar, porém vem sendo cada vez menos utilizado pelo fato de o tempo de entrega ser tipicamente longo. O modelo incremental realiza entregas de forma dependente, ou seja, uma versão básica é disponibilizada ao final do primeiro ciclo de desenvolvimento, e nos ciclos seguintes novas funcionalidades são agregadas, até que se tenha o produto completo. Já no modelo iterativo, uma versão básica de boa parte das funcionalidades é disponibilizada no primeiro ciclo e as funções melhoradas são disponibilizadas posteriormente.

Para cada uma destas categorias de ciclos de vida de *software* há um ou mais modelos formais disponíveis para adoção e os modelos mais conhecidos são:

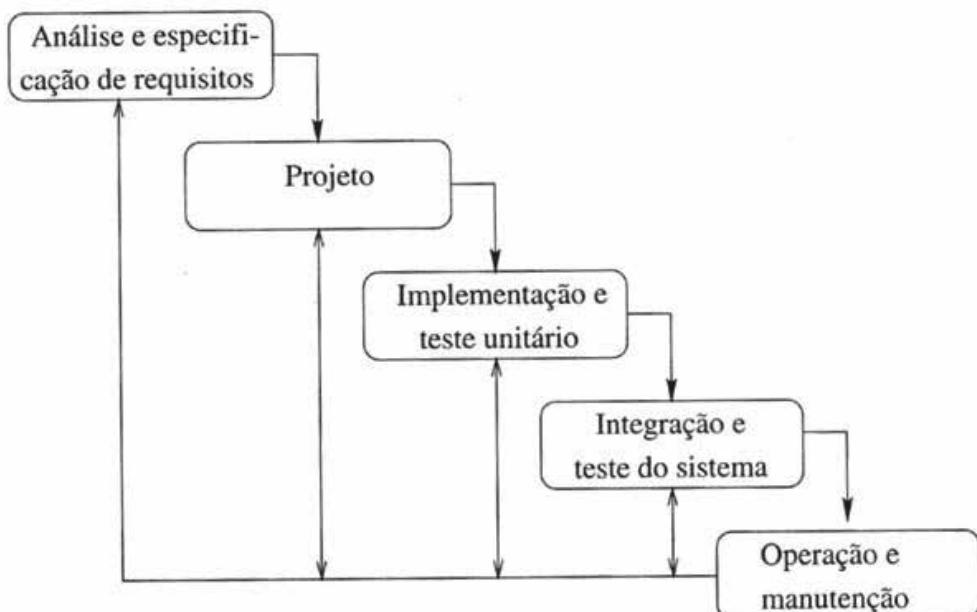
- O modelo Cascata ou sequencial.
- Modelo de Prototipação.
- Modelo Espiral.
- Modelo Iterativo e Incremental.
- Modelo Baseado em Componentes.
- Modelo em V.
- O Modelo RAD (*Rapid Application Development*).
- Modelo de Quarta Geração.

2.1 MODELO CASCATA OU SEQUENCIAL

O modelo cascata é um modelo de ciclo de vida clássico, pois trabalha numa abordagem sistemática em que as fases são estabelecidas pelas funções realizadas na engenharia convencional. Tem como característica marcante o fato de que todas as fases de desenvolvimento têm momentos de início e término bem definidos, sendo que uma fase só inicia se a anterior estiver concluída. Foi proposto por Winston W. Royce, no ano de 1970, e desde a década de 1980 é amplamente usado pela comunidade de desenvolvimento de *software*, pois é a mais conhecida, simples e fácil de trabalhar.

Conforme apresentado na Figura 17, ao final de cada fase ou marco de controle é feita uma revisão para avaliar se realmente pode-se avançar para a próxima e, quando apontar que o projeto não está apto a entrar na fase seguinte, ele permanece na fase corrente até que seja aprovado, ou seja, as atividades de especificação, codificação e testes seguem uma única disciplina rígida, com a qual nenhuma atividade se inicia sem que a anterior tenha sido encerrada e aprovada.

FIGURA 17 - CICLO DE VIDA CASCATA



FONTE: Disponível em: <<http://www.oficinadanet.com.br/artigo/gerencia/software-house-como-funciona-o-mercado-de-desenvolvimento>>. Acesso em: 18 jun. 2015

Embora o modelo cascata tenha fragilidades, ele é significativamente melhor do que uma abordagem casual de desenvolvimento de *software*, é considerado o modelo de fácil gestão, pelo fato de que suas etapas são bem definidas e sem sobreposição. É útil nos casos em que o domínio de aplicação é bem entendido e quando vários sistemas similares foram construídos anteriormente.

Como característica negativa deste modelo é a demora no atendimento devido aos retrabalhos tardios, ou seja, na maioria das vezes é praticamente impossível obter-se a totalidade de requisitos de maneira antecipada e em uma única etapa de um ciclo, por isso faz-se importante possibilitar o retorno para uma etapa sempre que for necessário. Como exemplo podemos citar quando os testes só ocorrerão no final de todo o processo, onde um erro sutil pode vir a exigir semanas de verificação para que se possa eliminá-lo, o que poderá impactar em atrasos no cronograma do projeto.

2.2 MODELOS POR PROTOTIPAÇÃO

As tecnologias de prototipação foram criadas no final dos anos 80 como alternativa a eliminar dificuldade de atender aos requisitos definidos pelos clientes e certificar-se de que efetivamente foram atendidas as suas necessidades.

A ideia deste modelo é produzir uma representação visual das funcionalidades que o *software* terá depois de pronto, permitindo avaliar as características antes que ele seja efetivamente desenvolvido e compreender com mais clareza como ficará sua usabilidade, trazendo maior velocidade de desenvolvimento e o envolvimento direto do usuário.

No modelo cascata se fazia um protótipo apenas como mecanismo para identificar ou auxiliar na identificação e elaboração dos requisitos finais do sistema e depois era descartado, mas por prototipação seu conceito tornou-se evolutivo, e o próprio protótipo passou a ser o produto entregue ao cliente.

Através deste modelo os desenvolvedores iniciam a especificação (requisitos) e codificação das partes essenciais do sistema em um protótipo, adicionando melhorias até se tornar o produto final. Sua aplicação permite a verificação da eficiência de determinadas rotinas ou fluxos de atividades, agilizando seu processo de desenvolvimento.

FIGURA 18 - MODELO DE PROTÓTIPO



FONTE: Disponível em: <<http://www.diegomacedo.com.br/modelos-de-ciclo-de-vida/?print=print>>. Acesso em: 19 jul. 2015

Importante destacar que este modelo é apropriado quando o cliente não obtém os requisitos de entradas e saídas devidamente definidos, quando o cliente participaativamente do projeto, construção e validação do protótipo através de uma metodologia ágil de desenvolvimento, a fim de se criar uma interface bem próxima daquela que o *software* irá possuir.

Durante as etapas de avaliação e refinamento do protótipo há *feedback* constante a respeito do produto, aumentando a visibilidade do que está sendo desenvolvido, ajudando nos casos em que os analistas e os clientes não conhecem bem o domínio da aplicação e o cliente está apressado para receber o projeto com urgência.



Para este tipo de ciclo de vida é preciso, antes, prevenir e orientar os clientes quanto à sua expectativa no prazo de entrega do projeto, pois acham que o protótipo já é o projeto pronto, e, neste modelo, se há muita dificuldade em se prever o tempo que será necessário para produzir um produto aceitável olhando somente para o protótipo. Sem levar em consideração requisitos de qualidade e sua manutenibilidade.

Existem diversas ferramentas para elaboração de protótipos de *software*, muitas linguagens de programação possuem a possibilidade de se fazer os protótipos do sistema, como, por exemplo, o Delphi e o Eclipse, porém outras, utilizadas pelos *designers*, como o Mockingbird, fazem os protótipos e depois só reaproveitam as telas para configurar e programar o restante do produto.

A utilização de prototipação é recomendada por diversos autores da área de Engenharia de *Software*, porém, a ideia básica deste modelo de processos de desenvolvimento de *software* é que o protótipo seja descartado e o processo construtivo siga as boas práticas de Engenharia de *Software*, evitando fragilidades da gestão de projeto e possa garantir semelhanças entre o protótipo e o *software* construído. Quando devidamente utilizado, reduzirá o retrabalho e atenderá rapidamente às expectativas dos usuários.



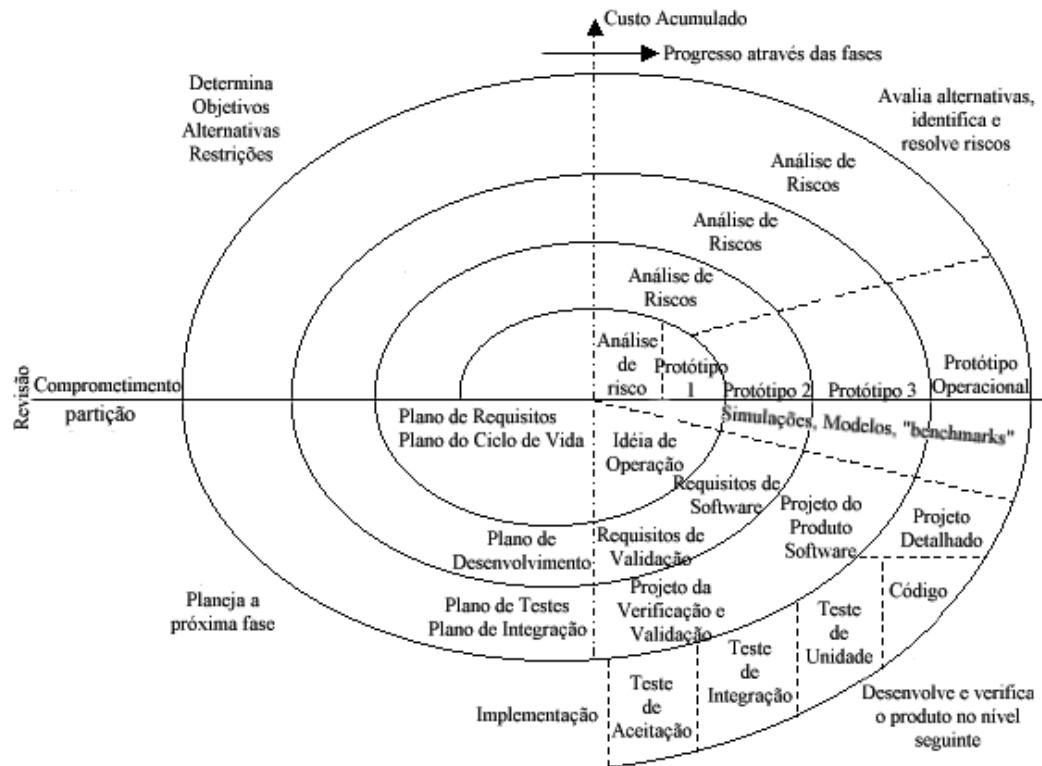
Um dos processos fundamentais durante o ciclo de vida de projeto de software é o levantamento de requisitos. Em muitos casos, as falhas estão na identificação correta das necessidades do cliente. No Tópico 4 desta Unidade 1 será apresentado sobre a área de Engenharia de Requisitos, disciplina extremamente importante para a engenharia de software.

2.3 MODELO ESPIRAL

O modelo espiral criado por Barry Boehm, no ano de 1988, traz uma abordagem das melhores características existentes nos modelos cascata e de prototipação vistos anteriormente, com ponto de vista orientado a riscos ao invés das abordagens orientadas à documentação e codificação.

Neste modelo, à medida que o desenvolvimento do *software* avança, percorre-se a espiral no sentido horário do centro para fora, incorporando inclusão de novos requisitos de forma evolutiva. Verifica-se que há sobreposição evolutiva durante quatro setores: primeiro, planejar os objetivos, as alternativas e restrições para tratamento dos riscos identificados; segundo, avaliação e redução de riscos, sua probabilidade e impacto durante cada fase do projeto; terceiro, com base nos riscos já identificados, desenvolve e valida estratégia de desenvolvimento para reduzir a probabilidade de ocorrência do risco; e quarto, realiza revisão e planejamento do projeto e decide pela continuidade ou não do próximo ciclo, seja durante as fases de elaboração dos requisitos, planos de implementação ou execução dos testes e integração do produto.

FIGURA 19 - MODELO ESPIRAL



FONTE: Disponível em: <<http://www.diegomacedo.com.br/modelos-de-ciclo-de-vida/?print-print>>. Acesso em: 19 jul. 2015

Uma consideração importante para este modelo é que seu foco está em aumentar a qualidade do planejamento em cada ciclo, dando maior visibilidade para a gerência, especialmente na análise de riscos e favorecendo o desenvolvimento de versões incrementais do *software*. Como exemplo de riscos que representem algum impacto no resultado final do *software*, podemos citar problemas devido à falta de segurança e performance, atraso de entrega do projeto, incompatibilidade de ferramentas, entre outros.

Kechi Hirama (2011) explica que o processo Espiral é dividido em uma série de atividades, chamadas regiões de tarefas, nas quais são realizadas as seguintes atividades:

- Comunicação do Cliente, que visa estabelecer uma comunicação efetiva entre desenvolvimento e cliente.
- Planejamento, em que se definem recursos, cronograma e outras informações de projeto.
- Análise de Riscos, na qual se avaliam os riscos técnicos e gerenciais.
- Engenharia, que consiste na implementação de uma ou mais representações do *software*.

- Construção e Entrega, quando se implementa, testa, instala o *software* e se provê apoio ao usuário.
- Avaliação do Cliente, que objetiva obter realimentação do cliente baseada na avaliação das representações do *software* implementadas durante as atividades de engenharia e construção e entrega.

FONTE: Hirama (2011, p. 210)

Portanto, devido ao nível de gerenciamento ser mais complexo, vai exigir a necessidade de maior experiência da equipe de desenvolvimento, sobretudo os responsáveis pela gerência, que terão que ter maior experiência da equipe na análise de riscos e maior esforço para o desenvolvimento, aumentando consideravelmente os custos do projeto.

2.4 MODELO ITERATIVO E INCREMENTAL

É um modelo que tem uma abordagem que divide o desenvolvimento de *software* em ciclos, sendo que o desenvolvimento evolui em versões de novas funcionalidades até que o sistema completo esteja construído. Cada ciclo é constituído pelas fases de análise, projeto, implementação e testes, já discutidas no Tópico 2 desta unidade.

Foi criado em resposta às fraquezas do modelo cascata. Adequado quando é possível dividir os requisitos do sistema em partes gradativas do desenvolvimento, em que sua alocação é realizada em função do grau de importância atribuído a cada requisito, onde cada ciclo considera um subconjunto de requisitos que produz um novo incremento ou iteração do sistema que contém extensões e refinamentos sobre o incremento ou iteração anterior.

O modelo de ciclo de vida Iterativo apresentado na Figura 20, a seguir, corresponde em desenvolver ou melhorar o projeto de *software* em etapas diferentes através de iterações. Pode-se citar como exemplo: um projeto possui três módulos diferentes e que são independentes um do outro, os quais poderão, identificando e especificando requisitos em iteração diferentes do projeto, ou seja, desenvolver na iteração 1 o módulo 1, iteração 2 o módulo 3 e na iteração 3 o módulo 2.

Este tipo de modelo é utilizado, especialmente, quando se faz necessário buscar estratégias diferentes de se executar o planejamento do projeto, por exemplo, quando se deseja liberar uma funcionalidade mais importante primeiro, quando se precisa alocar profissionais específicos em determinado período do desenvolvimento do projeto ou, até mesmo, pelos recursos financeiros destinados a um determinado momento daquele projeto.

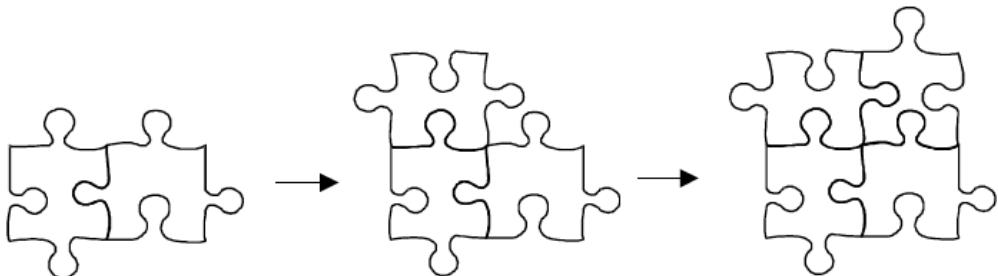
FIGURA 20 - MODELO DE CICLO DE VIDA ITERATIVO



FONTE: Disponível em: <http://wiki.sj.ifsc.edu.br/wiki/index.php/Ciclo_de_Vida_Iterativo_e_Incremental>. Acesso em: 19 jul. 2015

O modelo de ciclo de vida Incremental, apresentado na Figura 21, a seguir, demonstra que há dependência entre módulos de um sistema, portanto, para ser desenvolvido ou melhorado um segundo módulo do sistema, o primeiro módulo deverá estar 100% concluído e liberado para uso.

FIGURA 21 - MODELO DE CICLO DE VIDA INCREMENTAL



FONTE: Disponível em: <http://wiki.sj.ifsc.edu.br/wiki/index.php/Ciclo_de_Vida_Iterativo_e_Incremental>. Acesso em: 19 jul. 2015

Em cada ciclo ocorrem as atividades de análise de requisitos, projeto, implementação e teste, bem como a integração dos artefatos produzidos com os artefatos já existentes. Assim, o desenvolvimento evolui em versões, através da construção incremental e iterativa de novas funcionalidades, até que o sistema completo esteja construído.

Entre as principais vantagens dos modelos iterativo e incremental está a redução dos riscos envolvendo custos a um único incremento, pois, caso a equipe de desenvolvedores precisar repetir a iteração, a organização perde somente o esforço mal direcionado de uma iteração, não o valor de um produto inteiro.

O que conspira a favor deste tipo de modelo é a aceleração do tempo de desenvolvimento do projeto como um todo. Os desenvolvedores trabalham de maneira mais eficiente quando buscam resultados de escopo pequeno e claro, impedindo o risco de lançar o projeto no mercado fora da data planejada.

Outra evidência favorável a este tipo de ciclo de vida, que é muito comum hoje em dia nos projetos de *software*, é justamente que as necessidades dos usuários e os requisitos correspondentes não podem ser totalmente definidos no início do processo, precisando ser refinados em sucessivas iterações, possibilitando, com este modelo de operação, a facilidade de se adaptar a mudanças de requisitos.

Porém, o que conspira contra é que este modelo tem maior dificuldade de gerenciamento, pois as fases do ciclo ocorrem de forma simultânea, com amplo sincronismo entre atividades, pessoas, recursos etc., e também o cliente pode se entusiasmar excessivamente com a primeira versão do sistema e pensar que tal versão já corresponde ao sistema como um todo.



A abordagem incremental e iterativa somente é possível se existir um mecanismo para dividir os requisitos do sistema em partes, para que cada parte seja alocada a um ciclo de desenvolvimento. Essa alocação é realizada em função do grau de importância atribuído a cada requisito.

2.5 MODELO BASEADO EM COMPONENTES

A ideia de que o *software* deveria ser componentizado surgiu na conferência patrocinada pelo Comitê de Ciência da Organização do Tratado do Atlântico Norte (OTAN), que lançou o termo Engenharia de *Software*, na Alemanha, em 1969, devido à necessidade de agrupar coerentemente rotinas relacionadas de forma a montar componentes que poderiam ser reutilizados em diversos sistemas.

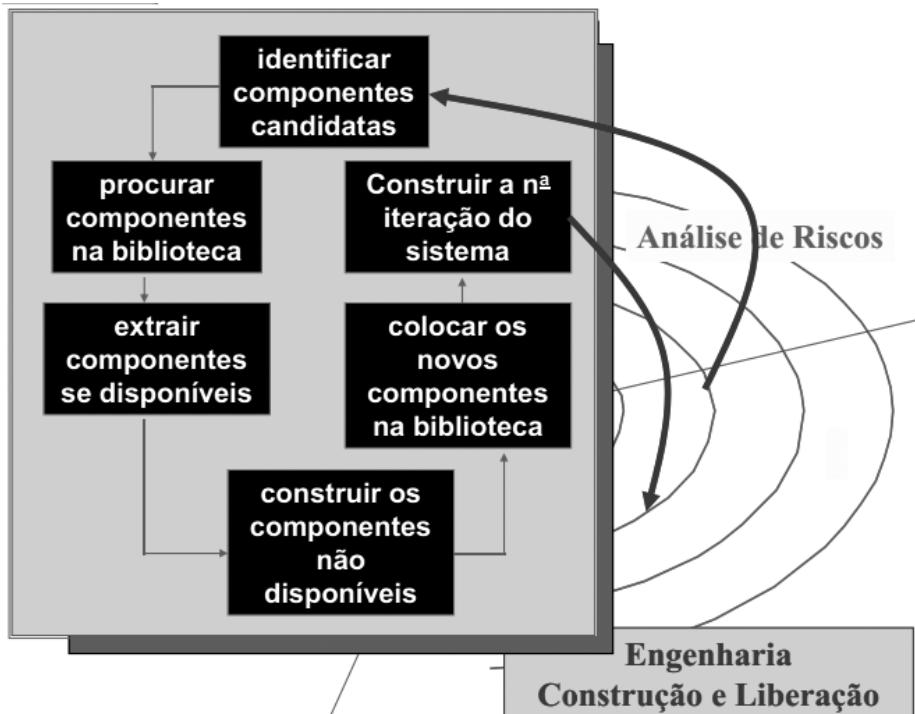
O desenvolvimento baseado em componentes depende da existência de uma grande biblioteca disponível de componentes de *software* reusáveis utilizados pela empresa. É ideal que se tenha algum *framework* de integração bem padronizado desses componentes, facilitando o processo de reúso de *software*. Este modelo aproveita bem o conceito do desenvolvimento espiral através de reúso informal e independentemente do processo de desenvolvimento utilizado.

Seu processo de funcionamento é relativamente fácil: a partir do momento em que subir o requisito do cliente, é feita uma busca na biblioteca de componentes pelo componente que trata sobre este requisito para implementação, extraíndo ele da biblioteca, fazendo a alteração (prática do reúso) e devolvendo este incremento na biblioteca deste *framework*; caso o componente já existir e estiver completo, faça o seu reúso. Porém, se não existir o componente candidato à alteração, deve-se criar um novo componente, desenvolvê-lo e fazer sua integração à biblioteca e liberar o produto pronto ao cliente. Esta reutilização possibilita a produção de programas a partir de partes já testadas e amplamente utilizadas.



Vale destacar que é necessário haver controle e organização desta biblioteca, para que não sejam perdidas novas versões dos componentes reusáveis.

FIGURA 22 - CICLO BASEADO EM COMPONENTES



FONTE: Disponíveis em: <<http://metodologiasclassicas.blogspot.com.br/p/modelos-especializados-de-processos.html>>. Acesso em: 19 jul. 2015

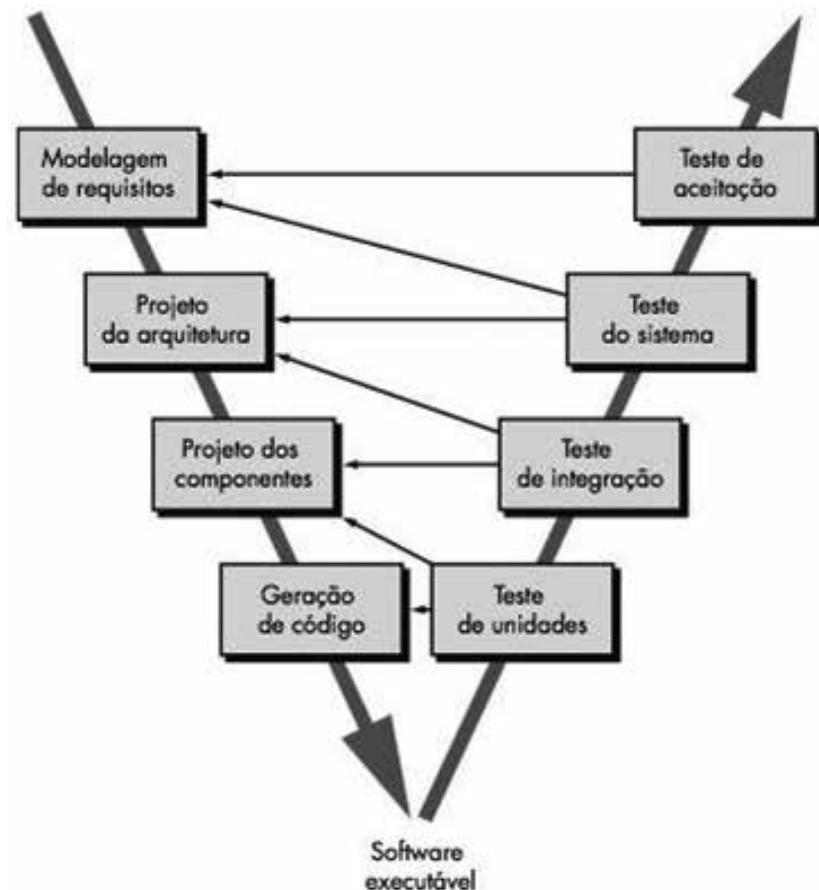
Dependendo da robustez tecnológica da biblioteca de componentes utilizada, os métodos de uma determinada classe podem ser reutilizáveis em diferentes aplicações e arquiteturas de sistema, oferecendo redução do tempo de desenvolvimento do custo do projeto. Estudos comprovam que esta redução de tempo e custo é bastante alta, já que o conceito de orientação a objetos permite o reúso de diferentes objetos, métodos dos componentes modelados.

Nota-se que as empresas que utilizam este modelo muitas vezes ficam presas pela tecnologia, não explorando novas soluções e impedindo, assim, atender às reais necessidades do usuário.

2.6 MODELO EM V

Este modelo enfatiza a estreita relação entre as atividades de teste com as demais atividades do processo de desenvolvimento de realizar a implementação do *software*. A análise e a especificação são realizadas com o apoio da área de testes, tomando como ponto de partida entender o problema e apresentar a melhor solução ao projeto. Parte-se da premissa de que, quanto mais tempo levar para entender, mais rápido se executa e menos erros são cometidos ao projeto.

FIGURA 23 - MODELO EM V



FONTE: Disponível em: <<http://www.devmedia.com.br/introducao-ao-modelo-cascata/29843>>. Acesso em: 19 jul. 2015

Neste modelo a ênfase está na decomposição de requisitos (lado esquerdo da figura) à integração dos testes (lado direito da figura). Os testes de unidade devem garantir que todos os aspectos detalhados na especificação do sistema foram codificados corretamente e, com os testes de integração, fazer a validação dos componentes utilizados. Quando a área da arquitetura passa a ser o foco, os testes de sistema validarão sua integração verificando se o sistema atende aos requisitos definidos na especificação técnica (projeto arquitetural) e, finalmente, os testes de aceitação pelo cliente buscam validar os requisitos para confirmar a homologação final do sistema.

Segundo Kichi Hirama (2011), as principais características do Modelo V são:

- Divisão entre atividades de desenvolvimento e de verificação e validação. As atividades de desenvolvimento são análogas às do processo cascata, porém a fase de testes é detalhada em testes de unidade, testes de integração, teste de validação e teste de sistema. Naturalmente, outros tipos de testes podem ser incluídos.
- Clareza nos objetivos de verificação e validação. Atividades de verificação significam responder à seguinte questão: “Estamos construindo certo o produto, ou seja, de maneira correta?”. No caso da validação a questão é: “Estamos construindo o produto certo, ou seja, o *software* é aquele acordado com o cliente?”. Os testes são instrumentos fundamentais para verificar e validar o *software* contra as suas especificações.
- Melhor planejamento dos testes. O fato de explicitar os tipos de testes e a sua correspondência com as atividades de desenvolvimento facilitam o planejamento ao especificar os testes e os recursos necessários.



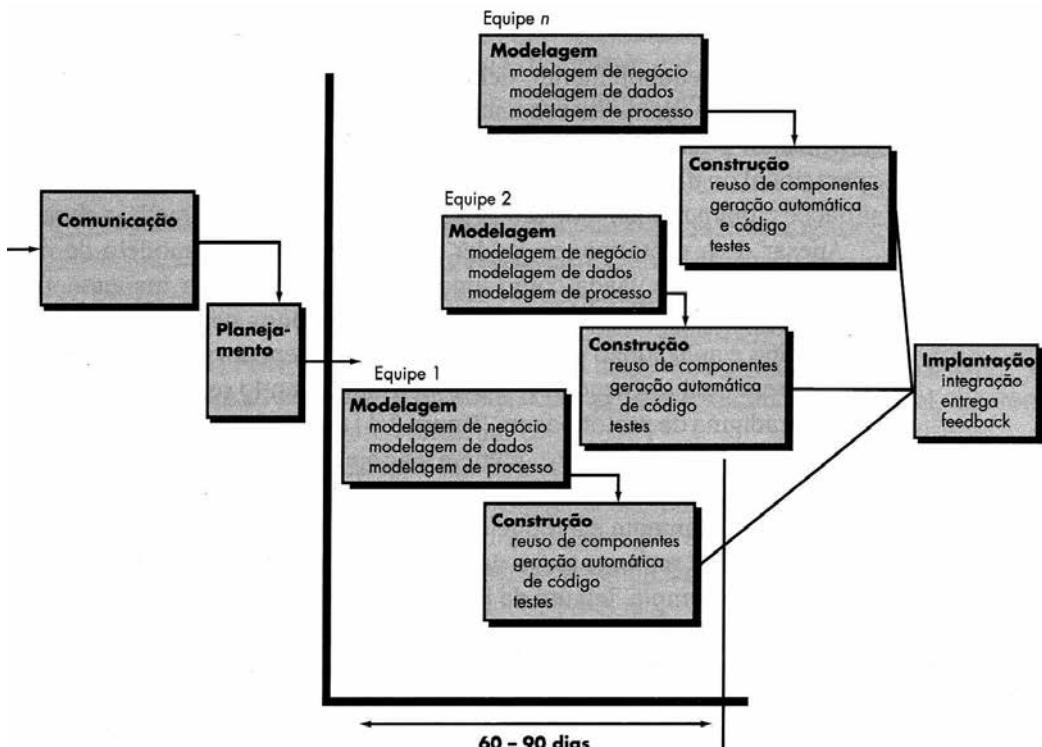
Maiores detalhes sobre a área de gerência de testes serão estudados na Unidade 2 deste Caderno de Estudos.

2.7 MODELO RAD (RAPID APPLICATION DEVELOPMENT)

O modelo de desenvolvimento rápido de aplicações foi criado na década de 1990 por James Martin, a fim de propor um processo incremental mais curto de desenvolvimento entre as etapas de modelagem e codificação. É um processo que prima por metodologias ágeis, de forma que é recomendado realizá-lo em até 90 dias.

O planejamento é essencial, porque várias equipes trabalham em paralelo em diferentes funções do sistema. De forma genérica, ele possui as fases de comunicação, planejamento, diversos incrementos durante a modelagem e construção enfatizada pelo uso de componentes de *software*, conforme visto no modelo anterior, e concluído na implantação do sistema.

FIGURA 24 - CICLO DE VIDA RAD



FONTE: Disponível em: <<http://adsbaixarengenhariadesoftware.blogspot.com.br/2013/05/introducao-engenharia-de-software.html>>. Acesso em: 23 jul. 2015

O modelo RAD traz como vantagem a diminuição com custos em alterações, pois os requisitos incompletos podem ser vistos e completados durante o desenvolvimento, deixando o *software* mais próximo da necessidade do cliente, melhorando sua manutenção. Este modelo não é adequado para qualquer tipo de *software* ou necessidade de aplicação. Recomenda-se que a empresa possua uma plataforma de desenvolvimento que privilegie a agilidade, ideal para ambientes que possam fazer uso de outros modelos ágeis, como no caso do desenvolvimento baseado em componentes ou classes preexistentes, como APIs (*Application Programming Interface* ou Interface de Programação de Aplicativos).

2.8 MODELO DE QUARTA GERAÇÃO

O modelo de quarta geração é baseado na utilização de ferramentas de alto nível próxima à linguagem natural para especificação e desenvolvimento de *softwares*. Há uma mudança de paradigmas da prototipação na forma do desenvolvimento de *software*, otimizando sua mão de obra. Pode abranger as ferramentas com possível facilidade ao gerenciamento de bancos de dados, geração em diferentes visões de relatórios, prototipação de interfaces, ferramentas de linguagem de programação inteligentes na geração de códigos, utilizando

recursos gráficos de alto nível e recursos da inteligência artificial explorados nos conceitos ontológicos e semânticos das aplicações.

Em geral, seu uso é limitado ao desenvolvimento de um universo restrito de sistemas, permitindo o desenvolvimento de produtos para domínio de aplicações muito específicos, pois se restringe pela dificuldade de utilização das ferramentas, ineficiência do código gerado e manutenibilidade questionável. Além disso, ainda não há ambientes que possam ser utilizados com igual facilidade para todas as categorias de *software*.



Além dos modelos de ciclo de vida descritos, encontramos na literatura muitos outros modelos, e cada um deles, geralmente, enfoca um determinado tipo de projeto específico, ou seja, depende do contexto do projeto onde ele será aplicado. Nenhum modelo de ciclo de vida é tido como ideal, pois aquele que é mais rápido e eficiente em certos projetos pode ser mais lento ou complexo quando o domínio do problema é trocado.

O mais importante é que as empresas de desenvolvimento de *software* tenham algum modelo de ciclo de vida, ou melhor, um "catálogo" de modelos personalizado de acordo com a sua estrutura organizacional e projetos (demandas e expectativas dos clientes) para que consigam níveis de excelência em seus produtos e serviços.

RESUMO DO TÓPICO 3

Neste tópico, você aprendeu que:

- Um ciclo de vida de desenvolvimento de *software* define o roteiro de trabalho do projeto, que é composto por fase, atividades e suas tarefas, utilizando métodos, técnicas, ferramentas e procedimentos na construção do produto de *software*.
- As fases do ciclo de vida são constituídas de planejamento (iniciação), análise e especificação de requisitos, projeto (elaboração), implementação (construção), testes, entrega e implantação (transição) e, por fim, operação e manutenção, passando desde a sua concepção até ficar sem uso algum após o término do projeto.
- Os modelos de processo de ciclo de vida de *software* podem ser linear, incremental ou iterativo, logo, compreendê-los poderá auxiliar na adoção de um dos modelos mais adequados à realidade e necessidade da organização.
- Modelo cascata ou sequencial possui um ciclo de vida clássico, em que todas as fases de desenvolvimento possuem início e fim bem definidos e não avança sem estiver com a fase anterior 100% concluída.
- O objetivo das tecnologias de prototipação é produzir uma representação visual das funcionalidades que o sistema terá depois de pronto, permitindo ao usuário e à equipe de desenvolvimento avaliar as características antes que ele seja efetivamente implementado e entregue.
- Inicialmente, o modelo de prototipação desenvolve uma visão da sua interface e depois reaproveita as telas para configurar e programar o restante do produto de *software*, trazendo maior rapidez na construção do projeto e aceitação dos usuários.
- O modelo espiral traz uma abordagem orientada à gestão de riscos ao invés de apenas orientar a documentação e codificação, como nos casos do modelo cascata e de prototipação.
- Também no modelo espiral, à medida que o projeto de *software* avança são incorporados novos requisitos de forma evolutiva, com sobreposição de atividades em cada fase do projeto, aumentando a qualidade do planejamento em cada ciclo e dando maior visibilidade à gerência.
- O modelo iterativo e incremental permite dividir o escopo do projeto em partes gradativas do desenvolvimento, evoluindo o projeto em versões de novas funcionalidades até o sistema estar completo.

- O modelo iterativo pode desenvolver os módulos ou funcionalidade de forma independente e liberar o projeto em partes; já no modelo incremental, poderá desenvolver em módulos, porém há dependência de funcionalidades entre estes módulos para um próximo módulo ser implementado, permitindo a liberação após o acréscimo de funcionalidades feitas nas iterações anteriores.
- O modelo baseado em componentes permite agrupar rotinas relacionadas de forma a montar componentes que podem ser reutilizados em diversos módulos do sistema.
- A partir do momento em que sobe o requisito do cliente, é feita uma busca na biblioteca de componentes, que trata sobre este requisito, e se já existir e estiver completo, faz-se o seu reuso; caso contrário, criar um novo componente e fazer sua integração à biblioteca, liberando o produto pronto ao cliente.
- O foco do modelo em V dá ênfase nas atividades de testes durante a análise, implementação e homologação do sistema, primeiramente validando-os antes do desenvolvimento, garantindo maior entendimento do problema e evitando enganos e retrabalhos.
- Modelo RAD (*Rapid Application Development*) propõe um ciclo de vida rápido de desenvolvimento utilizando um processo incremental entre as etapas de modelagem e codificação durante um período de até 90 dias.
- Já o modelo de quarta geração trata dos modelos de última geração, utilizando ferramentas de alto nível através da inteligência computacional, explorando os paradigmas da ontologia e semântica das aplicações, ou seja, próxima à linguagem natural.

AUTOATIVIDADE



- 1 Conceitue o que é um ciclo de vida de *software*.
- 2 Com relação aos quatro principais modelos de processo de ciclo de vida de *software*, preencha o nome correspondente:

O modelo _____ é apropriado quando o cliente não tem os requisitos de entradas e saídas devidamente definidos e o cliente participa ativamente na validação da sua interface.

No modelo _____ um *software* é desenvolvido em partes, cada qual adicionando alguma capacidade funcional a ele até que o *software* completo esteja implementado e a cada incremento são feitas extensões e modificações no projeto.

Modelo _____ é orientado a riscos, e suas atividades são apresentadas em uma dimensão radial, na qual há sobreposição evolutiva das atividades de especificação, projeto, desenvolvimento, homologação e implantação.

O modelo _____ se define como o desenvolvimento de um *software* e se dá de forma sequencial e linear a partir da atividade de verificação da viabilidade do desenvolvimento. Para cada etapa cumprida, segue-se a etapa imediatamente posterior.

De acordo com as sentenças acima, assinale a alternativa que traz a sequência correta das definições dadas para cada modelo:

- a) () Cascata – Iterativo e Incremental – Espiral - Modelo RAD.
 - b) () Prototipação – Iterativo Incremental – Espiral - Cascata.
 - c) () Espiral – Baseada em Componentes – Modelo em V - Cascata.
 - d) () Prototipação – Quarta Geração – Espiral - Cascata.
- 3 O modelo de ciclo de vida de um projeto de *software* é um aliado indispensável para o sucesso do projeto, permitindo aproveitar coerentemente seus recursos do projeto (pessoas, tecnologias, artefatos etc.). A respeito das características e importância dos modelos de ciclo de vida, assinale V para verdadeiro ou F para falso:
- () No modelo linear, o *software* é executado e entregue com todas as suas funcionalidades em apenas uma fase.
 - () O modelo iterativo realiza entregas de forma dependente, há outros iterações, ou seja, uma versão básica é disponibilizada ao final do primeiro ciclo de desenvolvimento e nos ciclos seguintes novas funcionalidades são agregadas, até que se tenha o produto completo para realizar a entrega.

- () Já no modelo incremental, uma versão básica de boa parte das funcionalidades é disponibilizada no primeiro ciclo e as funções melhoradas são disponibilizadas posteriormente.
- () O modelo em V enfatiza a estreita relação entre as atividades de teste com as demais atividades do projeto, verificando e validando todos os artefatos gerados desde o início até o fim.

A sequência correta é:

- a) () F – V – V – F.
- b) () V – F – V – F.
- c) () V – V – V – F.
- d) () V – F – F – V.

REQUISITO DE SOFTWARE

1 INTRODUÇÃO

A elaboração de um projeto, seja ele para criar um produto novo ou melhorá-lo, surge tipicamente em função de algum problema, oportunidade ou necessidade de negócio. A partir do momento em que for decidido iniciar um processo de construção de um *software*, deve-se definir o escopo do projeto através de uma lista de funcionalidades que se deseja disponibilizar para seus usuários no sistema, no qual estas necessidades identificadas são denominadas de requisito. Portanto, requisito é uma definição formal e detalhada de uma função do sistema.

Machado (2011) define requisitos como um conjunto de condições ou capacidades necessárias que um *software* deve possuir para que o usuário possa resolver um problema ou atingir um objetivo, ou para atender as necessidades ou restrições da organização ou dos outros componentes do sistema.

A análise e especificação dos requisitos têm vital importância no desenvolvimento de *softwares*, pois é nesta etapa da engenharia de *software* que são levantadas as informações de grande importância para a construção adequada do sistema (considerado marco decisivo de sucesso ou fracasso dos projetos).

A extração de requisitos é o processo de transformar o conhecimento tácito, que está na mente dos usuários, em conhecimento explícito via documentação formal. Essa transformação só é possível através da determinação dos objetivos do produto e das restrições para a sua operacionalidade, através de uma análise do problema e documentação dos resultados. A saída do processo de extração de requisitos é um documento de especificação do sistema que deve dizer o que o produto a ser desenvolvido deverá fazer, e não como deve ser feito.



Embora seja óbvio, é importante que se chame a atenção para o fato de que os requisitos são a base sobre a qual serão apoiadas todas as demais atividades do processo de *software*. Se os requisitos não forem devidamente especificados, o sucesso do projeto poderá estar comprometido desde seu início.

Estudos realizados por Arteiro (2015) mostram porque trabalhar com requisitos de *software* é tão importante. De acordo com um estudo do *Standish Group*, no ano de 2014, cinco dos oito principais fatores de falhas em projetos estão relacionados a requisitos (Tabela 1). Estes são: requisitos incompletos, baixo envolvimento do cliente, expectativas não realistas, mudanças nos requisitos e requisitos desnecessários.

TABELA 1 - PRINCIPAIS CAUSAS DE FALHAS EM PROJETOS DE SOFTWARE

Problema	%
Requisitos incompletos	13.1
Baixo envolvimento do cliente	10.6
Falta de recursos	12.4
Expectativa não realista	9.9
Falta de suporte gerencial	9.3
Mudanças nos requisitos	8.7
Falta de planejamento	8.1
Requisitos desnecessários	7.5

FONTE: Disponível em: <http://www.cin.ufpe.br/~in1020/arquivos/monografias/2013_2/iveruska.pdf>. Acesso em: 24 jul. 2015

Como o requisito é uma documentação que diz o que o *software* deverá realizar, o sucesso do projeto depende de sua definição clara, e a engenharia de requisitos é considerada uma etapa do processo de *software* muito importante, pois é extremamente árduo quando se elabora requisitos de *software* de forma vaga e incompleta, gerando retrabalho para a equipe do projeto.

Simpoi publicou em 2008 que 70% dos problemas encontrados em *softwares* são de especificação, ou da falta dela, sendo que apenas 60% das informações formais e informais estão documentadas. Uma das principais razões para isso

encontra-se no perfil dos desenvolvedores, de modo geral, definido por pessoas talentosas, que não têm interesse em seguir disciplinas ou normas, consideram seu trabalho como algo artístico e criativo, ao invés de algo sujeito a procedimentos de engenharia de requisitos.

Carvalho e Chiossi (2001) apontam algumas dificuldades no processo de obtenção de requisitos:

- falta de conhecimento do usuário das suas reais necessidades e do que o produto de *software* pode lhe oferecer;
- falta de conhecimento do desenvolvedor do domínio do problema;
- domínio do processo de extração de requisitos pelos desenvolvedores de *software*;
- comunicação inadequada entre desenvolvedores e usuários;
- dificuldade do usuário em tomar decisões;
- problemas de comportamento;
- questões técnicas.

FONTE: Carvalho e Chiossi (2001, p. 148)

Os requisitos de negócio são declarações, em uma linguagem natural, como diagramas de quais serviços são esperados do sistema e as restrições sob as quais ele deve operar; os requisitos de sistemas seriam os detalhamentos, as funções, os serviços e as restrições operacionais do sistema. O documento de requisitos de sistema (chamado de especificação funcional) necessita ser preciso e validado entre o cliente e o desenvolvedor. Este documento deve dizer exatamente como dever ser implementado.

Numa definição mais abrangente e não relacionada especificamente a alguma área de conhecimento, um requisito pode ser considerado como uma condição indispensável para alcançar determinado resultado. Essa definição, puramente linguística, ajuda a direcionar uma linha de raciocínio que justifica a importância dos requisitos nas atividades de criação de *softwares* (GIANESINI, 2008, p. 22).

De acordo com Lopes (2004), um requisito é uma característica do *software* necessária para o usuário solucionar um problema de forma a atingir um objetivo. Uma característica de *software* que deve ser realizada ou implementada por um sistema ou componente de sistema para satisfazer um contrato padrão, especificação ou outra documentação formalmente imposta.

Diversas atividades de requisitos de *software* ocorrem ao longo de todo o ciclo de vida do *software*, um trabalho que consiste na análise de requisitos para identificar, quantificar, definir, especificar, documentar, rastrear, priorizar e classificar os principais problemas que o futuro *software* deve resolver. A seguir serão apresentados os três tipos de requisitos de *software*.

1.1 REQUISITOS FUNCIONAIS

Para Sommerville (2011), os requisitos funcionais são declarações de serviços que o sistema deve fornecer, de como o sistema deve reagir a entradas específicas e de como o sistema deve se comportar em determinadas situações. Em alguns casos, os requisitos funcionais também podem explicitar o que o sistema não deve fazer.

Já segundo Martins (2010), os requisitos funcionais são aqueles que definem o comportamento do sistema, capturados por meio de casos de uso, que documentam as entradas, os processos e as saídas geradas.

1.2 REQUISITOS NÃO FUNCIONAIS

Sommerville (2011) explica que os requisitos não funcionais são restrições ao serviço ou funções disponíveis no sistema. Incluem restrições tecnológicas no processo de desenvolvimento e restrições impostas pelas normas. Ao contrário das características individuais ou serviços do sistema, os requisitos não funcionais, muitas vezes, aplicam-se ao sistema como um todo.

Martins (2010) define que os requisitos não funcionais são compostos por características não necessariamente associadas ao comportamento, com o objetivo de definir características do sistema conforme observadas pelo cliente, apontando o desenvolvimento na direção correta.

Sommerville (2011) menciona também que, na realidade, a distinção entre diferentes tipos de requisitos não é tão clara como sugerem estas definições simples. Um requisito de usuário relacionado com a proteção, tal como uma declaração de limitações de acesso a usuários autorizados, pode parecer um requisito não funcional. No entanto, quando desenvolvido em mais detalhes, esse requisito pode gerar outros requisitos, claramente funcionais, como necessidade de incluir recursos de autenticação de usuários no sistema.

A norma ISO/IEC 9126 (2015) caracteriza os requisitos não funcionais referentes à qualidade dos produtos usando os seguintes conjuntos:

- Funcionalidade: Adequação; Acurácia; Interoperabilidade; Segurança de acesso.
- Confiabilidade: Maturidade; Tolerância a falhas; Recuperabilidade.
- Usabilidade: Inteligibilidade; Apreensibilidade; Operacionalidade; Atratividade.
- Eficiência: Comportamento em relação ao tempo; Comportamento em relação aos recursos.
- Manutenibilidade: Analisabilidade; Modificabilidade; Estabilidade; Testabilidade.
- Portabilidade: Adaptabilidade; Capacidade para ser instalado; Coexistência; Capacidade de ser substituído.

1.3 REQUISITOS INVERSOS

Segundo Pinto (2012), estes são requisitos que o foco do projeto não contemplará. Reconhece-se a existência destes requisitos, mas será declarado

de forma explícita que o projeto não irá abordá-lo ou não entregará aqueles requisitos, sendo de vital importância para não causar expectativas por parte do cliente ou usuários, delimitando o escopo do projeto.

Gomedé (2011) comenta que estes requisitos definem estados e situações que nunca podem acontecer.

Pode-se dizer que os elementos da Engenharia de *Software* constituem o foco da gestão de requisitos. Não há a necessidade de que a equipe tenha experiência em gestão de requisitos, mas, se toda empresa que presta serviços de Engenharia de *Software* tiver uma célula de trabalho especializada em gestão de requisitos para dar suporte às áreas de interesse, sempre terá um ponto de partida para o correto planejamento do projeto e, tendo estes itens corretamente descritos e negociados, quase não existirá o risco de o sistema iniciar de forma errada e imprecisa.

Gerenciamento de requisitos é uma das atividades fundamentais ao processo de desenvolvimento de *software*. Constitui a base para a definição da arquitetura do sistema, para a implementação propriamente dita, para geração dos casos de testes e para validação do sistema junto ao usuário. Gerenciamento de requisitos está relacionado ao processo de controlar todo o processo de desenvolvimento tendo como referência a *baseline* de requisitos. Este processo visa manter planos, artefatos e atividades de desenvolvimento consistentes com o conjunto de requisitos definidos para o *software* (SAYÃO; BREITMAN, 2009, p. 1).

2 TÉCNICAS DE LEVANTAMENTO DE REQUISITOS

Observa-se que a maioria dos problemas no levantamento de requisitos está relacionada a não utilizar uma técnica adequada para extrair os requisitos do sistema, e o analista não os descreve de modo claro, conciso e consistente com todos os aspectos significativos do sistema proposto, pelo fato de, muitas vezes, o usuário principal do sistema não saber o que quer que o sistema faça ou sabe e não consegue transmitir para o analista.

A utilização de técnicas de levantamento de requisitos é recomendada por diversos autores de engenharia de requisitos e personalidades da área Engenharia de *Software*. Algumas das principais técnicas serão apresentadas a seguir, de forma resumida:

- *Brainstorming*: consiste em uma “tempestade de ideias”, sem julgamentos ou análises, que ocorre em um ambiente descontraído e informal. É ideal para buscar ideias de novos produtos.
- JAD: técnica utilizada para promover cooperação, entendimento e trabalho em grupo entre usuários e desenvolvedores.
- Análise de documentos quantitativos: consiste na análise de documentos, como formulários e relatórios.

- Reunião: é utilizada para licitação de requisitos em grupo, com o objetivo de promover o intercâmbio de ideias, sugestões e opiniões entre os participantes, visando à aceitação de um ponto de vista.
- Prototipagem: utilizado para atrair aspectos críticos quando se tem domínio mínimo da aplicação.
- Entrevista: conversa com o usuário para extrair tópicos importantes.
- Questionários: extraír respostas através de questões subjetivas e objetivas.
- Observação: observar o comportamento e o ambiente do indivíduo que toma decisões pode ser uma forma bastante eficaz de levantar informações que, tipicamente, passam despercebidas usando outras técnicas.
- Levantamento Orientado a Ponto de Vista: tem como objetivo captar pontos de vista dos usuários, analisar as diferenças e similaridades, formando, assim, o requisito do sistema.
- Etnografia: forma utilizada para entender a organização, sua cultura e o objetivo que o sistema deve alcançar.
- Caso de Uso: especifica um comportamento externo de um sistema descrevendo um conjunto de sequências de ações realizadas pelo sistema para produzir um resultado de valor observável por um ator, realizado através de uma interação típica entre um ator (usuário, outro sistema computacional ou um dispositivo) e um sistema.

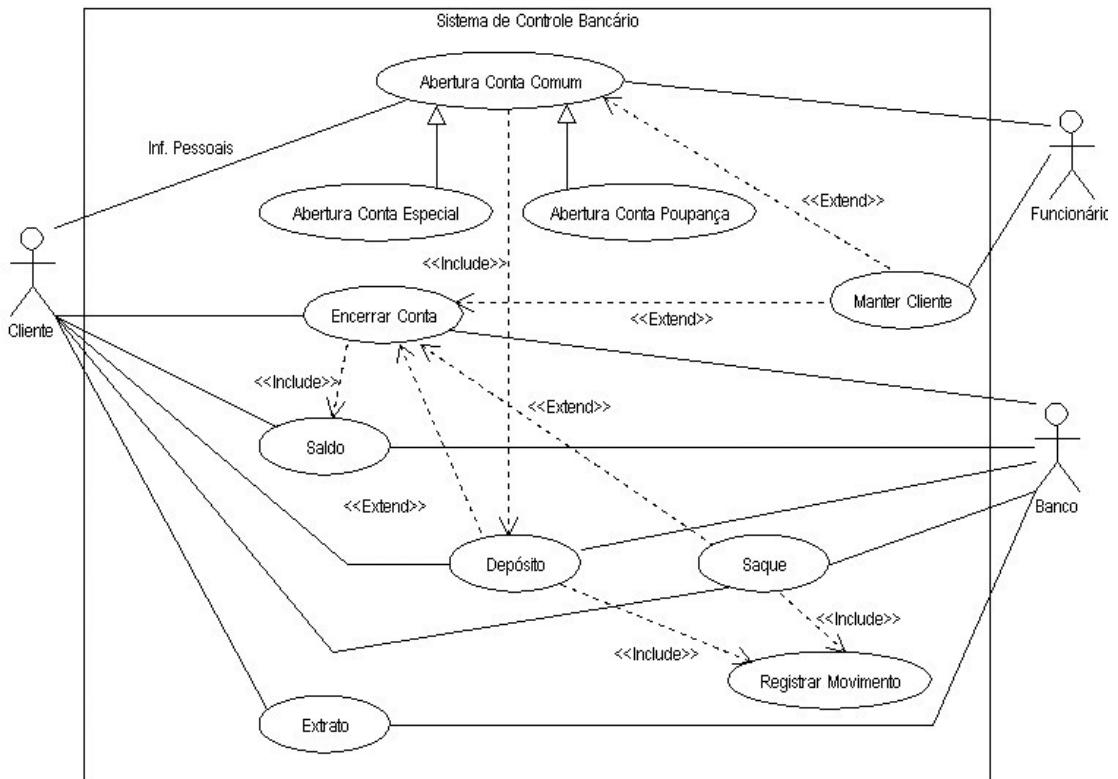


Todas as técnicas de levantamento de requisitos possuem vantagens e desvantagens a serem consideradas devido às inúmeras complexidades dos projetos, porém, a utilização de mais de uma técnica, de forma combinada, irá ajudar a melhorar a qualidade da completude dos requisitos.

Quando o cliente ou usuário interno ou externo de um projeto de *software* contrata a área de tecnologia, é preciso definir um bom atendimento destas necessidades ou desejos, então a área de tecnologia precisa elaborar, de uma forma clara, precisa e consistente, um documento com as definições mais detalhadas do sistema para que o cliente possa compreender e validar os requisitos.

Na fase de análise de requisitos, muitas organizações utilizam a modelagem para contextualizar um projeto. Para deixar mais clara esta ideia, os exemplos a seguir detalham requisitos de um “Sistema de Controle Bancário” extraídos do livro UML2. A figura detalha o diagrama de caso de uso que permite uma visão dos requisitos funcionais do sistema de forma rápida e objetiva.

FIGURA 25 - DIAGRAMA DE CASO DE USO



FONTE: Guedes (2011)

O quadro a seguir apresenta a documentação do requisito do caso de uso “abrir conta especial”, demonstrando forte relação do usuário com a funcionalidade do sistema. É recomendado que seja criada uma relação forte entre ambos, para que seja fácil de localizá-los e atualizá-los em caso de necessidade.

QUADRO 1 – DOCUMENTAÇÃO TEXTUAL DO CASO DE USO

Nome do CDU:	Abrir Conta Especial
CDU Geral:	Abrir Conta
Ator:	Funcionário
Resumo:	Este Caso de Uso descreve as etapas necessárias para a abertura de uma Conta Especial para um Cliente.
Pré-Condições:	O pedido de abertura deve ser aprovado.
Pós-condições:	É necessário realizar um depósito inicial.
Fluxo:	Passo 1: O funcionário solicita a abertura de Conta Especial. Passo 2: O funcionário consulta o cliente por seu CPF ou CNPJ. Passo 3: É definido o valor limite do cheque especial. Passo 4: É inserida uma senha de acesso. Passo 5: A conta é criada. Passo 6: É fornecido o valor a ser depositado. Passo 7: É realizado o registro do depósito. Passo 8: É emitido o cartão da conta.
Restrições / Validações:	Restrição 1: Para abrir uma conta especial é preciso ser maior de idade. Restrição 2: É necessário estar empregado e o salário ser superior a 500,00. Restrição 3: O valor mínimo de depósito inicial é R\$ 50,00.

FONTE: Guedes (2011)

3 DEFINIÇÕES E PRÁTICAS DE GERÊNCIA DE REQUISITOS PELAS PRINCIPAIS NORMAS DE DESENVOLVIMENTO DE SOFTWARE

Neste tópico serão abordados os principais conceitos relacionados às normas de desenvolvimento de *software*, com o objetivo de contextualizar a padronização do processo, tendo como referencial um conjunto de regras e diretrizes a serem adotadas para garantir a correta coleta e utilização dos requisitos.

3.1 CMMI

De acordo com Ananias (2009), o CMMI (*Capability Maturity Model Integration*), modelo reconhecido mundialmente por atestar a maturidade dos processos das organizações, em seu nível de maturidade dois – Gerenciado, possui uma área de processo chamada de Gestão de Requisitos, que tem o objetivo de satisfazer as necessidades das organizações em relação à forma de organizar os requisitos propostos pelo projeto e identificar inconsistências de acordo com o avanço dos projetos. Com isso, pode-se entender claramente o que deve ser feito e o que se espera obter como resultado.

3.1.1 Gerenciar requisitos

Para Hazan e Leite (2003), tem como objetivo principal o controle evolutivo dos requisitos, seja por deficiência ou novas necessidades dos requisitos registrados até o momento. Tem como objetivo gerenciar os requisitos e identificar inconsistências nos planos de projeto e produto.

3.1.1.1 Obter um entendimento dos requisitos

Ananias (2009) explica que esta prática visa desenvolver um entendimento com os fornecedores dos significados reais dos requisitos. À medida que o projeto amadurece e os requisitos vão sendo derivados, todas as atividades ou disciplinas receberão requisitos. Para impedir problemas futuros, critérios são estabelecidos para determinar canais apropriados ou fontes oficiais que devem recebê-los. A admissão de requisitos deve ser analisada com os responsáveis, no intuito de garantir um entendimento compatível e compartilhado sobre o significado dos requisitos. Esta análise e diálogo resultam em um acordo para assegurar o comprometimento entre os participantes do projeto e os atuais requisitos aprovados, com as alterações necessárias nos planos de projeto, atividades e produtos de trabalho.

3.1.1.2 Gerenciar mudanças de requisitos

Como dito por Ananias (2009), a terceira prática específica trata de gerenciar as mudanças nos requisitos à medida que evoluem durante o projeto, já que em seu andamento os requisitos mudam, de modo que podem ser incluídos, e as mudanças podem ocorrer nos já existentes. Faz-se necessário gerenciar as inclusões e mudanças de forma eficaz. A fonte de cada requisito deve ser conhecida e o fundamento lógico de qualquer mudança deve ser documentado, para que possa ser analisado de forma efetiva o impacto das alterações.

3.1.1.3 Manter rastreabilidade bidirecional dos requisitos

De acordo com o SEI, ou *Software Engineering Institute* (2015), quando se tem uma boa gestão de requisitos, a rastreabilidade pode ser estabelecida desde a fonte até o menor nível do requisito, e vice-versa. Esta rastreabilidade bidirecional ajuda a assegurar que os requisitos de origem foram tratados e se todos os níveis de requisitos podem ser rastreados até um requisito de origem.

3.1.1.4 Encontrar inconsistências entre trabalho de projeto e requisitos

Conforme o SEI (2015), com o objetivo de não ter inconsistências entre os planos de projeto, produtos de trabalho e os requisitos, tem-se necessidade de que tais inconsistências sejam identificadas, permitindo prover ações corretivas ao sistema, bem como documentar, incluindo o fundamento lógico, origens e condições.



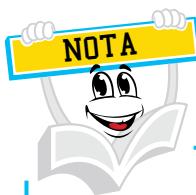
Na Unidade 3 deste Caderno de Estudos será abordado com maiores detalhes sobre o modelo CMMI.

3.2 MPS.BR

Segundo a Softex (2012), o MPS-BR é um programa mobilizador com o objetivo de melhorar o processo de software e serviços brasileiros, criado em dezembro de 2003, dividido em sete níveis de maturidade, e sua escala de maturidade se inicia no nível G (parcialmente gerenciado) e progride até o nível A (em otimização), estabelecendo patamares de evolução de processos, caracterizando estágios de melhoria da implementação de processos na organização.

Na abordagem do nível D (Largamente Definido), o processo de Gerência de Requisitos tem o propósito de gerenciar os requisitos dos produtos e componentes do produto, do projeto e identificar inconsistências entre requisitos, planos do projeto e os produtos de trabalho do projeto, com os seguintes resultados esperados (SOFTEX, 2012):

1. As necessidades, expectativas e restrições do cliente, tanto do produto quanto de suas interfaces, são identificadas.
2. Um conjunto definido de requisitos do cliente é especificado e priorizado a partir das necessidades, expectativas e restrições identificadas.
3. Um conjunto de requisitos funcionais e não funcionais do produto e dos componentes do produto que descrevem a solução do problema a ser resolvido é definido e mantido a partir dos requisitos do cliente.
4. Os requisitos funcionais e não funcionais de cada componente do produto são refinados, elaborados e alocados.
5. Interfaces internas e externas do produto e de cada componente do produto são definidas.
6. Conceitos operacionais e cenários são desenvolvidos.
7. Os requisitos são analisados, usando critérios definidos para balancear as necessidades dos interessados com as restrições existentes.
8. Os requisitos são validados.



Na Unidade 3 deste Caderno de Estudos será abordado com maiores detalhes o modelo MPS.Br.

4 GERENCIAR MUDANÇAS DE REQUISITOS

Um projeto de desenvolvimento de *software* produz muitas informações e o gerenciamento de requisitos trata não somente de entregas do *software* ao cliente, mas das mudanças que podem ocorrer nos requisitos em que o rastreamento dos requisitos permite analisar os impactos dessas mudanças durante o desenvolvimento de *software*.

As mudanças devem ser gerenciadas para assegurar que atendam a questões econômicas e contribuam para a necessidade de negócio da organização que adquire o sistema. Mais importante do que discutir as causas é saber como minimizar os impactos das mudanças, as mudanças de requisitos devem ser gerenciadas para que sejam aprovadas, revisadas e efetivamente implementadas, por meio de rastreamento de requisitos, análise de impactos e gerenciamento de configurações.

O gerenciamento de configuração tem por objetivo garantir que todos os elementos que constituem o projeto do *software* estejam devidamente identificados, controlados e versionados. Estes elementos do projeto são todos os artefatos produzidos durante o processo de construção do *software*, logo, envolvem documentação, modelagem, código, casos de teste, planos de projeto etc.



Na Unidade 2 deste Caderno de Estudos será abordada com maior detalhe a área gerência de configuração de *software*. Porém, a gestão das mudanças destes artefatos é tratada pela gerência de requisitos.

De acordo com Hirama (2011), a análise de impactos de mudanças de requisitos tem por objetivo identificar de que forma uma mudança impacta nos planos do projeto, que contêm as estimativas aprovadas de esforço e custo para os produtos de trabalho e tarefas, bem como os códigos de unidade ou módulos de *software* que necessitam ser modificados.

Enfatiza este mesmo autor que o rastreamento de requisitos está relacionado com a recuperação de fontes dos requisitos e o prognóstico dos efeitos dos requisitos, de uso fundamental para a análise de impacto quando os requisitos mudam.

Segundo o IEEE (1988), a rastreabilidade é o grau em que o relacionamento pode ser estabelecido entre dois ou mais produtos do processo de desenvolvimento,

especialmente produtos que tenham um relacionamento predecessor-sucessor ou mestre-subordinado do outro; por exemplo, o grau em que requisitos e projeto de um dado componente de *software* combinam.

Um importante mecanismo para gerenciar mudanças é através de uma matriz de rastreabilidade, que permite registrar o relacionamento entre dois ou mais produtos de desenvolvimento. Como exemplo podemos apresentar a figura a seguir, na qual há dependência entre requisitos funcionais e entre requisitos não funcionais de um determinado projeto. O preenchimento de “x” na matriz significa que, quando for alterado o requisito funcional RF01, ele poderá impactar nos requisitos funcionais RF3 e RF4 e nos requisitos não funcionais RNF02 e RNF03. O preenchimento do “◊” na matriz significa que, quando alterar o RF03, ele poderá impactar nos requisitos funcionais RF01, RF04 e RF05 e nos requisitos RNF01, RNF02 e RNF04.

TABELA 2 - MATRIZ DE RASTREABILIDADE DE REQUISITOS DE UM DETERMINADO PROJETO

Requisito Funcional	RF01	RF02	RF03	RF04	RF05
RF01			X	X	
RF02					
RF03	◊			◊	◊
Requisito Não Funcional					
RNF01			◊		
RNF02	X		◊		
RNF03	X				
RNF04			◊		

FONTE: O autor

O uso da matriz de rastreabilidade do exemplo acima permitirá que o impacto da mudança de um único requisito gerará alteração e testes somente nele mesmo ou se deve alterá-lo e testar também outros requisitos integrados com ele. Pode acontecer de alterar um só requisito e testá-lo e outros deste módulo do sistema ou, até mesmo, alterar e testar os outros devido às suas dependências e regras de negócio.

Um projeto de desenvolvimento de *software* não é isento a mudanças. Toda mudança é particularmente danosa para os objetivos de um projeto. É necessário, então, ter uma estratégia diferente de gerenciamento de projeto que possa cuidar exclusivamente da evolução dos requisitos. Nem sempre uma mudança é viável, mas a análise de impactos baseada em registros de uma matriz de rastreabilidade permite argumentar com os clientes sobre as consequências no processo de desenvolvimento de *software* e nos custos do projeto.

RESUMO DO TÓPICO 4

Neste tópico, você aprendeu que:

- A engenharia de requisitos é responsável por definir a lista de funcionalidades que irão compor o escopo de um projeto e seu sucesso dependerá de como estas necessidades são levantadas formalmente.
- A análise e a especificação de requisitos são fundamentais em um projeto de *software*, mapeando o conhecimento tácito em conhecimento explícito, identificando as regras de negócio dizendo o que o sistema deverá fazer.
- Existem duas categorias de requisitos: os requisitos de negócio, que irão detalhar quais serviços e restrições são esperados do sistema, e os requisitos de sistemas, que irão detalhar as funções e restrições operacionais do sistema, a primeira atividade executada pelo analista de negócios e a segunda pelo analista de sistemas.
- Os requisitos de negócio ou de sistemas podem ser realizados por três tipos: os requisitos funcionais, que definem o comportamento do sistema; os requisitos não funcionais, que incluem restrições impostas pelas normas ou tecnologias; e os requisitos inversos, que irão informar tudo o que não deverá contemplar no sistema.
- Vimos, também, que as principais técnicas de levantamento de requisitos são: *brainstorming*, JAD, análise de documentos quantitativos, reunião, prototipagem, entrevista, questionários, observação, levantamento orientado a ponto de vista, etnografia e casos de uso. Cada uma utilizando estratégias diferentes para extrair da melhor forma possível as necessidades dos usuários ou clientes.
- O modelo CMMI (*Capability Maturity Model Integration*) propõe como práticas de gerência de requisitos de *software*, em seu nível 2 de maturidade, as seguintes atividades: obter um entendimento dos requisitos, gerenciar mudanças de requisitos, manter rastreabilidade dos requisitos e controlar inconsistências entre trabalho de projeto e requisitos.
- Já o MPS.Br propõe, em seu nível D de maturidade, o processo de Gerência de Requisitos, com o propósito de gerenciar os requisitos dos produtos e componentes do produto, do projeto, e identificar inconsistências entre requisitos, planos do projeto e os produtos de trabalho do projeto.

- É inevitável ocorrerem mudanças e estas devem ser gerenciadas através das mudanças de requisitos para minimizar seus impactos, com sua respectiva aprovação, revisão por meio de rastreamento de requisitos, análise de impactos e gerenciamento de configurações para depois serem implementadas.
- A matriz de rastreabilidade é um recurso importante para saber os impactos e dimensão que uma alteração pode provocar em outros requisitos ou funcionalidades. Ela permite argumentar com os clientes sobre as consequências no processo de desenvolvimento de *software* e nos custos do projeto.
- Atualmente, não há uma definição clara de quais são os melhores ou mais adequados métodos ou técnicas para gerência de requisitos no desenvolvimento de *software*, sendo assim, a maioria das organizações utiliza métodos próprios de tratar e suprir esta carência.

AUTOATIVIDADE



1 Trabalhar com requisito de *software* é muito importante, porém muitas organizações falham quando estão elaborando seus projetos. Assinale a opção correta dos fatores de falhas da área de requisitos segundo estudos do Standish Group (2014):

- () Falta de recursos tecnológicos e humanos - requisitos desnecessários - falta de suporte gerencial - especificação de sistema incompleta - mudança de requisitos.
- () Requisitos incompletos - expectativas não realistas - baixo envolvimento da gerência - falta de planejamento - falta de modelagem.
- () Requisitos incompletos - baixo envolvimento do cliente - expectativas não realistas - mudanças nos requisitos - requisitos desnecessários.
- () Falta de suporte gerencial - falta de recursos tecnológicos e humanos - baixo envolvimento do cliente - mudanças nos requisitos - requisitos incompletos.

2 Com relação aos três tipos de requisitos de *software*, preencha as lacunas a seguir:

Os _____ são declarações de funcionalidades do sistema, de como o sistema deve reagir a entradas específicas de dados.

Os _____ incluem restrições tecnológicas no processo de desenvolvimento e restrições impostas pelas normas.

Os _____ definem estados e situações que nunca podem acontecer delimitando o escopo do projeto.

De acordo com as sentenças acima, marque a alternativa que traz a sequência CORRETA das definições dadas para cada requisito.

- a) () Requisitos não funcionais - requisitos funcionais - requisitos inversos.
- b) () Requisitos inversos - requisitos não funcionais - requisitos funcionais.
- c) () Requisitos inversos - requisitos funcionais - requisitos não funcionais.
- d) () Requisitos funcionais - requisitos não funcionais - requisitos inversos.

3 Existem diversas técnicas a serem aplicadas para o levantamento de requisitos, de modo que possam identificar claramente os problemas identificados pelos usuários. A respeito destas técnicas, assinale V para verdadeiro ou F para falso.

- () Prototipagem: utilizada para promover cooperação, entendimento e trabalho em grupo entre usuários e desenvolvedores.
- () Entrevista: conversa com o usuário para extrair tópicos importantes.
- () *Brainstorming*: é ideal para buscar ideias de novos produtos em um ambiente descontraído e informal.

- () Observação: utilizado para atrair aspectos críticos quando se tem domínio mínimo da aplicação.
- () Caso de Uso: descreve um conjunto de sequências de ações realizadas pelo sistema para produzir um resultado de valor realizado através de uma interação típica entre um ator e um sistema.

A sequência CORRETA é:

- a) () F – V – F – F – V.
- b) () V – F – V – F – F.
- c) () F – V – V – F – V.
- d) () V – F – F – V – V.

- 4 Quais atividades-padrão os modelos CMMI e MPS.Br recomendam para gerenciar requisitos de *software*?
- 5 Descreva quais são as ações necessárias para gerenciar mudanças de requisitos de *software*.
- 6 Como funciona uma matriz de rastreabilidade?

UNIDADE 2

GERENCIAMENTO DE PROJETOS DE SOFTWARE, ESTIMATIVAS E MÉTRICAS DE PROJETOS DE SOFTWARE E GERÊNCIA DE CONFIGURAÇÃO DE SOFTWARE

OBJETIVOS DE APRENDIZAGEM

Ao final desta unidade, você será capaz de:

- compreender os principais conceitos sobre Gerenciamento de Projeto de Software;
- entender como se devem realizar as estimativas e métricas de Projetos de Software;
- conhecer a importância da Gerência de Configuração de Software.

PLANO DE ESTUDOS

Esta unidade de ensino contém três tópicos, sendo que no final de cada um, você encontrará atividades que contribuirão para a apropriação dos conteúdos.

TÓPICO 1 – GERENCIAMENTO DE PROJETOS DE SOFTWARE

TÓPICO 2 – ESTIMATIVAS E MÉTRICAS DE PROJETOS DE SOFTWARE

TÓPICO 3 – GERÊNCIA DE CONFIGURAÇÃO DE SOFTWARE

CERENCIAMENTO DE PROJETOS DE SOFTWARE

1 INTRODUÇÃO

Independente do ramo de atuação em que as organizações estão inseridas, o mercado está se tornando cada vez mais complexo, exigindo mudanças rápidas e fazendo com que novas estratégias de forma consciente, ou mesmo inconsciente, sejam desenvolvidas no intuito de melhorar o desempenho no gerenciamento de projetos de *Software*.

Um projeto é composto por várias etapas, e cada etapa deve ser muito bem administrada. Em um mundo globalizado, onde a concorrência é cada vez mais acirrada, a gestão desses empreendimentos de forma eficiente pode representar um diferencial competitivo. Para que isso seja possível, investimentos com projetos se torna algo fundamental para obter êxito exigindo-se conhecimento de planejamento, bem como técnicas e ferramentas para colocar em prática uma gestão eficaz.

O gerente de projetos deve ser uma pessoa bastante capacitada, atendendo aos requisitos necessários, para que fique à frente do projeto e consiga efetuar uma boa gerência, evitando possíveis erros.

Para que um projeto de *software* seja bem-sucedido, é necessário que alguns parâmetros sejam corretamente analisados, como por exemplo, o escopo do *software*, os riscos envolvidos, os recursos necessários, as tarefas a serem realizadas, os indicadores a serem acompanhados, os esforços e custos aplicados e a sistemática a ser seguida (PRESSMAN, 1995).

2 DEFINIÇÃO DE PROJETO

Segundo o site do *Project Management Institute – PMI* (2015), projeto é um conjunto de atividades temporárias no sentido de que tem um início e fim definidos no tempo, realizadas em grupo, destinadas a produzir um produto, serviço ou resultado único.

Além de definir o projeto como algo temporário, com sequência de atividades com início, meio e fim, Carvalho e Rabechini Jr. (2011) ressaltam que seu resultado final fornece um produto ou serviço único e progressivo, tangível ou intangível limitado a restrições de tempo e custo.

Por maior que seja o tempo de seu desenvolvimento, um projeto não dura para sempre. Terá fim assim que seus objetivos sejam alcançados ou quando se chegar à conclusão de que estes objetivos não serão ou não poderão mais ser atingidos. Embora um projeto seja temporário, seu resultado tende a ser duradouro ou permanente, podendo ser classificados também como resultados tangíveis ou intangíveis.

Um projeto é único, pois mesmo que já tenha sido executado anteriormente ao ser colocado em prática novamente, o cenário envolvido (necessidades, pessoas, tecnologias) já mudou. A realidade do projeto atual já é outra em decorrência das mudanças ocorridas.

É considerado como progressivo, pois à medida que temos mais conhecimento sobre ele, vamos elaborando-o progressivamente, melhorando o seu detalhamento e as particularidades que o definem como único. Mas, podemos dizer que a característica de destaque de um projeto é o risco, pois nunca podemos ter a certeza de que o mesmo será bem-sucedido.

Vale destacar que projetos das mais variadas naturezas enfrentam os mesmos problemas ao redor do mundo. Entre os principais problemas que os projetos enfrentam, podemos citar: atrasos nos prazos, estouro de orçamento, falta de qualidade nas entregas, falta de motivação da equipe e ocorrência de eventos inesperados ao longo da execução do projeto.

Cada projeto possui um ciclo de vida, que ajuda a definir o início e término de cada etapa, o que deve ser realizado e por quem deve ser executado (matriz de responsabilidade do projeto). Serve para dar alicerce ao tripé de sucesso dos projetos: TEMPO/CUSTO/ESCOPO. A entrega do escopo deve ser feita no prazo estipulado, dentro do orçamento apontado, com nível de qualidade atendendo às necessidades do cliente comprador (VARGAS, 2009).

FIGURA 26 – TRIPÉ DE SUCESSO DOS PROJETOS



FONTE: Disponível em: <<http://www.gp3.com.br/images/restricoes.png>>. Acesso em: 15 ago. 2015.

3 GERÊNCIA DE PROJETOS

“O gerenciamento de projeto de *software* é uma atividade de apoio da engenharia de *software*. Inicia-se antes de qualquer atividade técnica e prossegue ao longo da modelagem, construção e utilização do *software*”. (PRESSMANN, 2011, p. 15).

Gerência de projetos é um conjunto de práticas que serve de guia a um grupo para trabalhar de maneira produtiva. Ela compreende métodos e ferramentas que organizam as tarefas, identificam sua sequência de execução e dependências existentes, apoia a alocação de recursos e tempo, além de permitir o rastreamento da execução das atividades e medição do progresso relativo ao que foi definido no plano de projeto (DA SILVA FILHO, 2015).

Gerenciamento de projeto envolve planejamento, monitoração e controle de pessoas, processos e eventos que ocorrem à medida que o *software* evolui desde os conceitos preliminares até sua disponibilização completa. (PRESSMAN, 2011).

O plano de projeto é um documento essencial que orienta o gerente de projeto, sendo obrigatório seu desenvolvimento e manutenção. Ele define os marcos de projeto e as principais atividades necessárias à sua execução. É bom planejar minuciosamente o projeto, pois a maioria dos insucessos ocorre devido à falta de planejamento ou à sua inexistência.

A gerência do projeto, de certa forma, tem responsabilidade com todos os envolvidos no projeto, porém isso pode variar de um projeto a outro. Geralmente um engenheiro de *software* gerencia suas atividades diárias, planejando e monitorando as atividades técnicas. Os gerenciadores de projeto planejam e controlam o trabalho da equipe de engenheiros de *software*. E o gerente do projeto coordena essa interface entre os profissionais do *software*.

As pessoas devem ser organizadas para o trabalho de desenvolvimento de forma efetiva, e a comunicação com o cliente deve ser bem compreendida, assim como o projeto deve ser planejado com prazos e metas estabelecidas. Por isso, Pressman (2011) define que o gerenciamento de projeto no desenvolvimento de *software* tem um foco nos 4 Ps: pessoas, produto, processo e projeto:

- **Pessoal:** Este recurso foca na formação da equipe, comunicação, ambiente de trabalho, desenvolvimento de carreira, análise da competência e cultura de equipe.
- **Produto:** Antes mesmo de começar o projeto, deve-se estabelecer os objetivos do produto e considerar as soluções. Somente desta forma é que se pode estimar custos, avaliar riscos e propor um cronograma.
- **Processo:** O processo do *software* é que vai fornecer a metodologia por meio da qual um plano de projeto abrangente para o desenvolvimento de *software* pode ser estabelecido. Poucas são as atividades metodológicas aplicáveis a todos os projetos.

- **Projeto:** Os projetos são planejados e controlados minuciosamente, pois esta é a maneira de administrar sua complexidade e garantir qualidade.

Devido à dinamicidade e competitividade do mercado, diversas organizações têm buscado alternativas tecnológicas para otimizar seus processos, diminuir custos e maximizar lucros, melhorar a qualidade de seus serviços e produtos, com o objetivo de atingir ou manter-se em um patamar importante no mercado em que atuam. E por este motivo é relevante a aplicação das práticas do PMI e PMBOK neste tipo de projeto.

Mas o que significam essas siglas? PMI é a sigla para *Project Management Institute* ou Instituto para Gerenciamento de Projetos. É uma entidade internacional sem fins lucrativos que foi fundada em 1969 nos EUA e reúne profissionais da área de gerenciamento de projetos. O PMI publica periodicamente diversos livros, mas o mais conhecido é o PMBOK. O *Project Management Body of Knowledge* é um conjunto das melhores práticas em gerenciamento de projetos, redigido através da colaboração de especialistas das mais diversas áreas. Atualmente, o PMI e o PMBOK são, respectivamente, a organização mais reconhecida e o manual mais utilizado na gerência de projetos em todo o mundo.



Maiores informações a respeito do PMI e PMBOK encontram-se no Tópico 8 desta unidade.

4 SUBPROJETOS, PROGRAMAS E PORTFÓLIO

Nem mesmo os melhores gerentes de projeto conseguem gerenciar sem compreender o ambiente onde se está inserido. Isso significa ter a percepção que seu projeto é uma parte importante no trabalho e entender que a compreensão da estratégia geral adotada pela organização contribui para um melhor gerenciamento do mesmo. Neste contexto os projetos podem ser compostos por três tipos de categorias: subprojetos, programas e portfólio.

FIGURA 27 – SUBPROJETOS, PROGRAMAS E PORTFÓLIOS



FONTE: Disponível em: <<https://apostilaadministrativa.wordpress.com/2012/08/01/conceitos-basicos-de-gerenciamento-de-projetos/>>. Acesso em: 15 ago. 2015.

Projetos grandes e de alta complexidade podem ser particionados em projetos menores facilitando assim a atividade de gerenciamento. Por exemplo podemos citar a existência de vários subprojetos para fabricação de peças específicas em cada subprojeto de um modelo de carro.

Já programas são grupos de projetos que são relacionados e gerenciados coletivamente de forma coordenada. Um exemplo de programa seria um novo sistema de satélite de comunicação com projetos para o design do satélite e das estações terrestres, construção de cada uma delas, integração do sistema e lançamento do satélite.

E o portfólio é um conjunto de projetos ou programas com objetivos afins. Um exemplo de portfólio de projetos seria o conjunto de projetos para atingir o objetivo de uma organização em reduzir a rotatividade de seus colaboradores, podendo surgir projetos associados a diferentes áreas de uma organização: Finanças, RH, Operações, *Marketing* etc. No nosso exemplo, teríamos como possíveis projetos: criação de programas de bolsa de estudo para funcionários, projeto para implantação de *Home Office*, projeto para criação de creches, projeto para revisão da política de premiação por desempenho, projeto para implantação de política de PLR (Participação nos Lucros e Resultados), projeto para divulgação do atual Plano de Carreira, e outros. Outro exemplo, em uma empresa de desenvolvimento de *software*, o conjunto dos projetos do departamento de novos produtos e do departamento de manutenção pode ser considerado como o portfólio da empresa.

FIGURA 28 – DIFERENÇA ENTRE PROJETOS, PROGRAMAS E PORTFÓLIO

Projetos	Programas	Portfólio
O sucesso é medido pelo orçamento, prazo e produtos entregues dentro das especializações	O sucesso é medido em termos de retorno do investimento (ROI), novas habilidades e benefícios entregues	O sucesso é medido em termos da performance agregada dos componentes do portfólio
O sucesso é medido pelo orçamento, prazo e produtos entregues dentro das especializações	O estilo de liderança é focado na gestão dos relacionamentos e solução de conflitos	O estilo de liderança é focado na adição de valor à tomada de decisão no portfólio
Gerentes de Projetos gerenciam técnicas, especialistas, etc	Gerentes de Programas gerenciam gerentes de projetos	Gerentes de Portfólio podem gerenciar ou coordenar a equipe de gerenciamento de portfólio
Gerentes de Projetos conduzem planejamento detalhado para gerenciar a entrega dos produtos do projeto	Gerentes de Programas criam planos de alto nível (não detalhados) para fornecer diretrizes para os projetos, cujos planos de detalhes serão criados posteriormente	Gerentes de Portfólio criam e mantêm os processos necessários e as comunicações para agregar o portfólio

FONTE: Trentin (2011, p. 20)

5 FASES DA GERÊNCIA DE PROJETOS

A gestão de projetos é composta por cinco fases: conceitual, definição, produção, operacional e encerramento.

- **Fase conceitual:** o estudo de viabilidade, onde realiza o desenvolvimento da ideia do projeto, a análise dos aspectos técnicos, custo e prazo e a definição de ambientes e da avaliação dos objetivos do projeto.
- **Fase de definição:** definição do plano do projeto, com seu custo, prazo, expectativas de desempenho técnico e a avaliação dos resultados do projeto antes que sejam empenhados grandes recursos em seu desenvolvimento.
- **Fase de produção:** período onde os resultados do projeto são produzidos e entregues como um produto, serviço ou processo organizacional efetivo.
- **Fase operacional:** a partir do momento que o produto existir e considerado viável em termos econômicos e práticos, valendo a pena a sua implementação e fornecendo um *feedback* do planejamento realizado.
- **Fase de encerramento:** se obtém a aceitação final do cliente, o gerente do projeto irá avaliar e relatar as lições aprendidas na execução do projeto. A seguir, um modelo genérico de ciclo de vida de um projeto:

FIGURA 29 – MODELO GENÉRICO DE CICLO DE VIDA DE UM PROJETO



FONTE: O autor

6 ESTRUTURA ORGANIZACIONAL

O *Project Management Body of Knowledge* – PMBOK (2014) classifica as organizações em diferentes estruturas de acordo com o grau de aderência às práticas sugeridas para a gerência de projetos.

FIGURA 30 – ESTRUTURA ORGANIZACIONAL

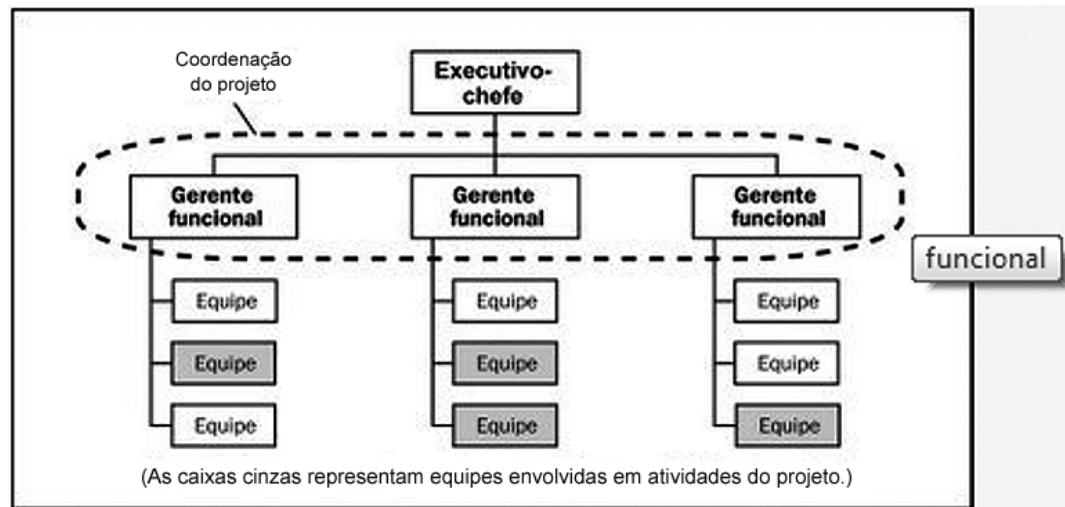


FONTE: O autor

As organizações funcionais ou hierárquicas são aquelas onde praticamente todas as decisões de gerenciamento de projetos são tomadas com auxílio dos gerentes de departamento, também chamados de gerentes funcionais. Nesse tipo de organização, os gerentes funcionais têm o controle dos recursos e do orçamento do projeto, cabendo aos gerentes de projeto a realização de tarefas administrativas.

Nesta estrutura a orientação é vertical, onde as ordens vêm de cima para baixo, sendo inadequado para projetos especializados, que devem ser gerenciados horizontalmente.

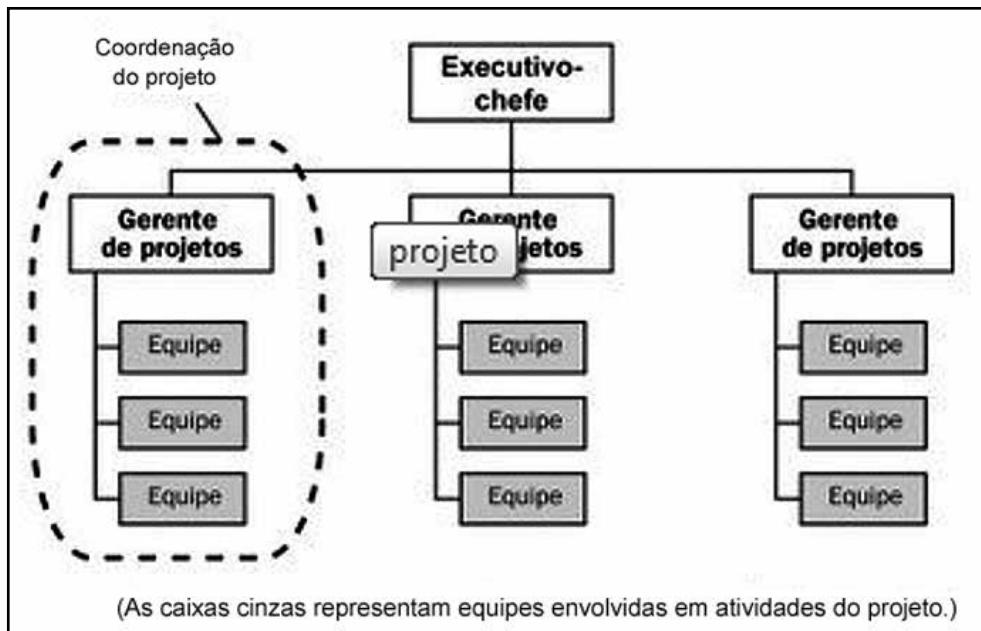
FIGURA 31 – ORGANIZAÇÃO FUNCIONAL



FONTE: Disponível em: <<https://brasil.pmi.org>>. Acesso em: 15 ago. 2015.

Quando somente o Gerente de Projetos toma as decisões referentes ao projeto, realiza os controles sobre os recursos e o orçamento do projeto. Uma desvantagem é a alocação e realocação das pessoas que compõem as equipes, uma vez que os projetos são temporários.

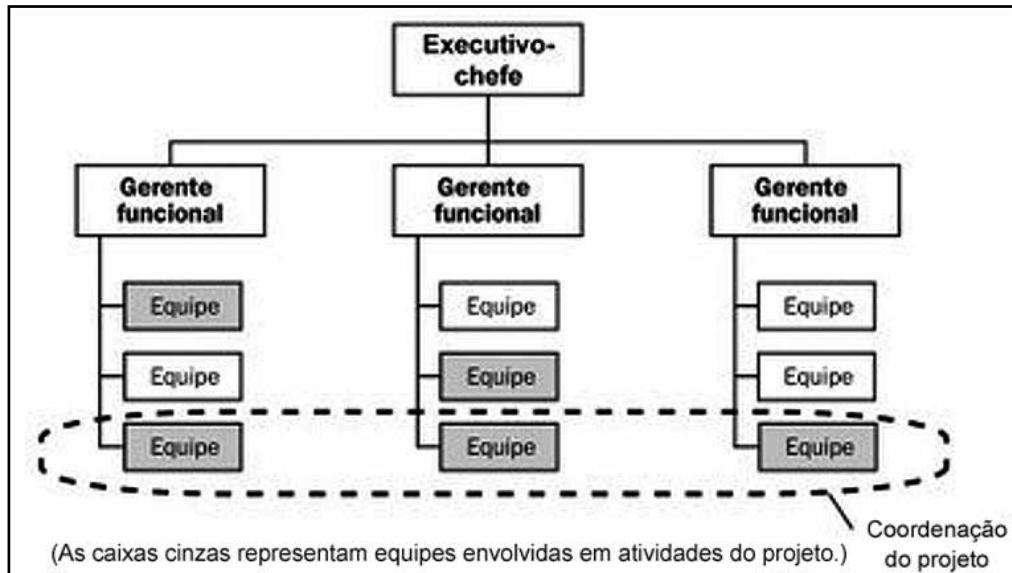
FIGURA 32 – ORGANIZAÇÃO PROJETIZADA



FONTE: Disponível em: <<https://brasil.pmi.org>>. Acesso em: 15 ago. 2015.

As organizações matriciais ficam entre os dois extremos, dividindo-se em matriz fraca, balanceada ou forte. Quanto maior a autoridade do GP e maior o grau de aderência às práticas do PMBOK, mais à direita se enquadra a organização.

FIGURA 33 – ORGANIZAÇÃO MATRICIAL



FONTE: Disponível em: <<https://brasil.pmi.org>>. Acesso em: 15 ago. 2015.

A organização matricial ainda pode ser dividida em:

- Matricial fraca: é uma estrutura muito similar com uma organização funcional e o papel do gerente de projetos é mais parecido com a de um coordenador ou facilitador. As prioridades das atividades do projeto são baixas tendo em vista que as equipes são subordinadas aos seus respectivos departamentos de origem.
- Matriz balanceada: nesta estrutura já se reconhece a necessidade de um gerente de projetos, porém com pouca autonomia sobre o projeto e seu financiamento já que ainda está sob a gestão de um gerente funcional. Numa estrutura com este modelo destaca-se o aumento considerável dos conflitos em função de que o superior não é claramente definido.
- Matricial forte: é uma estrutura que se assemelha com as características de uma organização projetizada e neste caso podem ter gerentes de projetos em tempo integral com autoridade considerável e pessoal administrativo trabalhando para o projeto em tempo integral. Neste modelo, as equipes estão mais comprometidas com o projeto e a prioridade dos projetos dentro da organização é alta.

FONTE: Disponível em: <<http://daseuspulos.com.br/2013/12/14/entenda-os-tipos-de-estruturas-organizacionais>>. Acesso em: 15 ago. 2015.

FIGURA 34 – INFLUÊNCIAS ORGANIZACIONAIS NOS PROJETOS

Influências organizacionais nos projetos

Características do projeto	Estrutura da organização	Funcional	Matriz			Projetizada
			Matriz fraca	Matriz balanceada	Matriz forte	
Autoridade do gerente de projetos	Pouca ou nenhuma	Limitada	Baixa a moderada	Moderada a alta	Alta a quase total	
Disponibilidade de recursos	Pouca ou nenhuma	Limitada	Baixa a moderada	Moderada a alta	Alta a quase total	
Quem controla o orçamento do projeto	Gerente funcional	Gerente funcional	Misto	Gerente de projetos	Gerente de projetos	
Papel do gerente de projetos	Tempo parcial	Tempo parcial	Tempo integral	Tempo integral	Tempo integral	
Equipe administrativa de gerenciamento de projetos	Tempo parcial	Tempo parcial	Tempo parcial	Tempo integral	Tempo integral	

FONTE: Disponível em: <<http://projetoseti.com.br/a-estrutura-de-sua-empresa-influencia-as-suas-atividades-diarias/>>. Acesso em: 16 ago. 2015

7 GERÊNCIA DE RISCOS

A missão da gerência de riscos é identificar riscos e buscar mecanismos que possam atenuar ou, até mesmo, eliminar os riscos do projeto. A análise e gestão de risco são ações que ajudam uma equipe de *software* a entender e gerenciar a incerteza. Independentemente do resultado, é aconselhável identificá-lo, avaliar sua probabilidade de ocorrência, estimar seu impacto e estabelecer um plano de contingência caso o problema realmente ocorra.

Quem faz a gerência de riscos são todos os envolvidos na gestão de qualidade: gerentes, engenheiros de *software* e demais interessados. A análise é muito importante, pois evita que estes erros ocorram, sendo este um elemento primordial no gerenciamento de projeto de *software*.

Nascimento (2003) destaca que os riscos podem ser de três naturezas: riscos de projeto, de produto ou de negócios.

- Riscos de projeto: eles ameaçam o plano do projeto, isto é, se os riscos do projeto se tornarem reais, é possível que o cronograma fique atrasado e os custos aumentem. Os riscos de projeto identificam problemas potenciais de orçamento, cronograma, pessoal, recursos, clientes, e requisitos e seu impacto sobre o projeto de *software*.
- Riscos técnicos: ameaçam a qualidade e a data de entrega do *software* a ser produzido. Se um risco técnico potencial se torna realidade, a implementação pode se tornar difícil ou impossível. Os riscos técnicos identificam problemas potenciais de projeto, implementação, interface, verificação e manutenção. Além disso, a ambiguidade de especificações, a incerteza técnica, a obsoléncia

técnica e a tecnologia “de ponta” também são fatores de risco. Riscos técnicos ocorrem porque o problema é mais difícil de resolver do que se pensava.

- Riscos de negócio: eles ameaçam a viabilidade do *software* a ser criado e muitas vezes ameaçam o projeto ou o produto. Os candidatos aos cinco principais riscos de negócio são: (1) criar um excelente projeto ou produto ou sistema que ninguém realmente quer (risco de mercado); (2) criar um produto que não se encaixe mais na estratégia geral de negócios da empresa (risco estratégico); (3) criar um produto que a equipe de vendas não sabe como vender (risco de vendas); (4) perda de suporte da alta gerência devido à mudanças de profissionais (risco gerencial); e (5) perda do orçamento ou do comprometimento dos profissionais (riscos de orçamento).

A gestão de riscos, compreende resumidamente quatro etapas:

- Identificação de riscos onde se busca identificar riscos de projeto ou de negócios. Para cada risco identificado, associa-se uma magnitude de risco que serve para indicar o grau de severidade e, portanto, de prioridade de tratamento do risco.
- Análise de risco que visa obter a probabilidade de ocorrência do risco e correspondente impacto sobre o projeto. Note que o impacto pode ser o atraso no projeto, que é tanto indesejável ao cliente quanto implica em custo adicional.
- Administração de risco que tem duas metas: (a) desenvolver uma estratégia de controle que serve para mitigar ou reduzir o impacto de um risco, e (b) elaborar um plano de contingência o qual recomenda as decisões (alternativas) a serem tomadas caso o risco aconteça.
- Monitoração de risco que faz uso de indicadores com o objetivo de monitorar e detectar a ocorrência ou probabilidade de ocorrência de um risco.

FONTE: Gestão de Projeto de Software. Disponível em: <<http://www.devmmedia.com.br/gestao-de-projetos-de-software/9143>>. Acesso em: 16 ago. 2015.

Um risco é definido pelo PMBOK como um evento que, em caso de ocorrência, terá efeitos negativos ou positivos para pelo menos um dos objetivos do projeto. Nesse contexto, o principal objetivo da gerência de riscos é diminuir a probabilidade e impacto dos eventos negativos e aumentar a probabilidade e impacto dos eventos positivos.

Uma maneira eficiente de se identificar e registrar os riscos é fazer uma Estrutura Analítica dos Riscos (EAR), com o objetivo de categorizá-los. A EAR é uma estrutura hierárquica muito semelhante à Estrutura Analítica e Projeto (EAP) utilizada durante o planejamento do projeto, só que você a utiliza em riscos ao invés de pacotes de trabalho. Em projetos de tecnologia da informação, normalmente são utilizadas três categorias básicas: Riscos Técnicos, Riscos Externos e Riscos Organizacionais. Em seus projetos você pode partir destas três e criar mais categorias conforme a sua necessidade.

Para identificar os impactos e as probabilidades de ocorrência dos riscos, normalmente são utilizadas a Estrutura Analítica dos Riscos, a declaração de escopo do projeto, o caminho crítico e a opinião de especialistas. A priorização destes riscos, com base nos impactos e na probabilidade de ocorrência, pode ser feita através de uma matriz de riscos que consiste em uma estrutura onde se faz a multiplicação entre um valor dado para a probabilidade de ocorrência de um risco e o valor dado para o impacto que a ocorrência desse risco traria para o projeto.

Uma vez concluída a priorização dos riscos, deve-se definir a forma de tratamento. Existem basicamente quatro formas de se tratar um risco (ARAÚJO, 2015).

- 1) Eliminação: é a melhor alternativa, quando existe essa possibilidade. Por exemplo, em um projeto de implantação de rede você percebe que a rede cai constantemente devido a um modem defeituoso utilizado para fazer conexão com a internet. Nesse caso, a troca do modem resolveria o problema e o risco, causado pelo modem, estaria eliminado.
- 2) Mitigação: ocorre quando não existe a possibilidade de eliminação do risco e tenta-se reduzir o impacto de ocorrência do mesmo. Em nosso projeto de implantação de rede, digamos que não existe a possibilidade de trocar o modem. Nesse caso, seria interessante deixar um serviço automatizado para avisar o técnico responsável pelo modem assim que a Internet caísse e deixar esse mesmo técnico mais próximo do modem, para que o conserto ocorresse o mais rápido possível.
- 3) Transferência: ocorre quando se paga para outra pessoa lidar com o risco para você. Em nosso exemplo, poderíamos ter pago uma garantia estendida para o dispositivo, de forma a podermos trocá-lo sem custo em caso de defeito por um período mais longo.
- 4) Aceitação: situações onde não existe a possibilidade dos tratamentos anteriores. Quando esse tipo de risco existe em um projeto, o mais importante é ficar ciente das consequências.

Para encerrar, devemos entender que é impossível se preparar para todos os riscos do projeto. Sempre existirão riscos imprevisíveis que devem ser tratados em caso de ocorrência. Para tentar identificar e tratar esses riscos ao longo da execução do projeto existe o processo Monitorar e Controlar os Riscos, que procura atualizar o plano de gerenciamento de riscos continuamente.

8 GESTÃO DE PROJETOS: PMI E O PMBOK

O *Project Management Institute – PMI* (2015) é uma instituição sem fins lucrativos, criada em 1969, que tem como principal objetivo contribuir para melhoria contínua da gestão de projetos. O PMI tem sido responsável por catalogar e divulgar o conhecimento e práticas de gestão de projetos visando melhorar a taxa de sucesso de execução de projetos e assim melhor capacitar os profissionais envolvidos, os gerentes de projeto.

No início da década de 90 foi publicada a primeira edição do Guia PMBOK (*Project Management Body Of Knowledge*) que se tornou o pilar básico para a gestão

de direção de projetos. O objetivo principal do Guia PMBOK é identificar o subconjunto do conjunto de conhecimentos em gerenciamento de projetos que é amplamente reconhecido como boa prática. Em outras palavras, definir e mostrar de forma geral práticas aplicáveis à maioria dos projetos na maior parte do tempo.



O Guia PMBOK também fornece um vocabulário comum para se discutir, escrever e aplicar o gerenciamento de projetos. O PMI utiliza o PMBOK como base, mas não como a única referência de gerenciamento de projetos para seus programas de desenvolvimento profissional.

De acordo com o PMBOK (2013), um gerenciamento de projetos eficaz exige que a equipe de gerenciamento de projetos entenda e use o conhecimento e as habilidades de pelo menos cinco áreas de especialização.

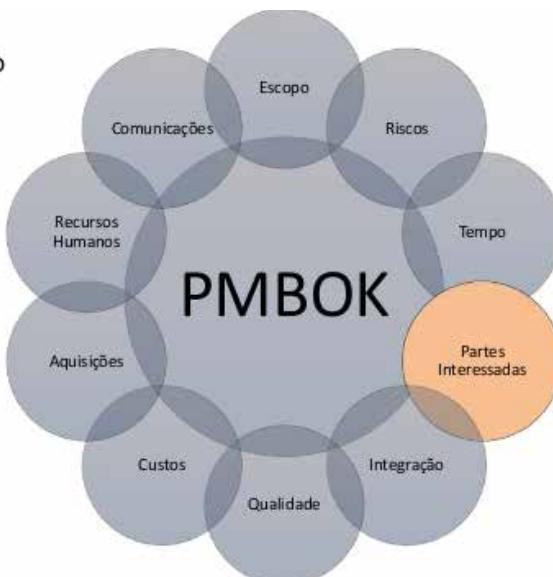
- Conjunto de conhecimentos em gerenciamento de projetos: descreve o conhecimento exclusivo da área de gerenciamento de projetos.
- Conhecimento, normas e regulamentos da área de aplicação: são categorias de projetos que possuem elementos comuns significativos nesses projetos, mas não são necessárias ou estão presentes em todos os projetos. As áreas de aplicação são geralmente definidas em termos de: departamentos funcionais e disciplinas de apoio, elementos técnicos, especializações em gerenciamento e setores.
- Entendimento do ambiente de projeto: praticamente todos os projetos são planejados e implementados em um contexto social, econômico e ambiental e têm impactos intencionais e não intencionais positivos e/ou negativos. A equipe deve considerar o projeto em seus contextos ambientais cultural, social, internacional, político e físico.
- Conhecimento e habilidades de gerenciamento geral: inclui o planejamento, a organização, a formação pessoal, a execução e o controle de operações de uma empresa existente. Ele inclui disciplinas de apoio como: contabilidade e gerenciamento financeiro, compras e aquisições, vendas e *marketing*, contratos e legislação comercial, fabricação e distribuição, logística e cadeia de abastecimento, planejamento estratégico, tático e operacional, estruturas organizacionais, comportamento organizacional, administração de pessoal, compensação, benefícios, planos de carreira, práticas de saúde e segurança e tecnologia de informação.
- Habilidades interpessoais: o gerenciamento de relações interpessoais trata da comunicação eficaz, influência sobre a organização, liderança, motivação, negociação e gerenciamento de conflitos e resolução de problemas.

A 5^a edição do Guia PMBOK aborda normas individuais para a gerência dos projetos definindo conceitos associados à gestão dos mesmos. A versão atual considera 47 processos divididos em 5 grupos de processos e 10 áreas de conhecimento que são comuns em quase todos os segmentos de negócios e seus respectivos projetos.

FIGURA 35 – PROCESSOS E ÁREAS DE CONHECIMENTO DO PMBOK, 5^a EDIÇÃO, 2013

- O guia PMBOK 5^a Edição possui **47 processos de gerenciamento**.
- Esses processos são agrupados em **10 áreas de conhecimento** distintas.

! Uma área de conhecimento representa um conjunto completo de conceitos, termos e atividades que compõem um campo profissional, campo de gerenciamento de projetos, ou uma área de especialização



FONTE: Disponível em: <<https://brasil.pmi.org>>. Acesso em: 16 ago. 2015.

Lembramos que TODOS os 47 processos do PMBOK se utilizam de entradas, ferramentas e técnicas e saídas. Os processos geralmente têm entradas distintas, mas podem existir casos onde uma mesma entrada serve para diversos processos. Outro detalhe que se deve ter em mente é que os processos geralmente acontecem em sequência. Isso ocorre porque é muito comum a saída de um processo servir como entrada para outro. Por exemplo: o processo de desenvolver o plano de gerenciamento do projeto só pode começar quando o processo gerar termo de abertura do projeto encerrar, pois a saída do segundo, o termo de abertura, é essencial como entrada no primeiro.

As dez áreas de conhecimento do PMBOK (5^a edição) estão distribuídas da seguinte forma:

- Gerenciamento/Gestão de integração do projeto.
- Gerenciamento/Gestão do escopo do projeto.
- Gerenciamento/Gestão de tempo do projeto.
- Gerenciamento/Gestão de custos do projeto.
- Gerenciamento/Gestão da qualidade do projeto.
- Gerenciamento/Gestão de recursos humanos do projeto.

- Gerenciamento/Gestão das comunicações do projeto.
- Gerenciamento/Gestão de riscos do projeto.
- Gerenciamento/Gestão de aquisições do projeto.
- Gerenciamento/Gestão de envolvidos do projeto (**adicionada na 5ª edição**).

A seguir, o conjunto de conhecimentos PMBOK 4^a e 5^a edição:

1. Gerenciamento de integração do projeto – trata sobre como as diversas áreas de conhecimento organizacional interagem entre si dentro de um projeto, ou seja, descreve os processos requeridos para certificar-se de que os vários elementos do projeto estão propriamente coordenados. Consiste em realizar as seguintes tarefas:
 1. Desenvolver o termo de abertura do projeto.
 2. Desenvolver o plano de gerenciamento do projeto.
 3. Orientar e gerenciar a execução projeto.
 4. Monitorar e controlar o trabalho do projeto.
 5. Executar o controle integrado de mudanças.
 6. Encerrar o projeto ou fase.
2. Gerenciamento do escopo do projeto – está relacionado com os limites do produto ou serviço que será construído com o projeto. Aborda atividades necessárias para que as entregas do projeto tenham sucesso, levantando os requisitos para definir o que deve ser feito no universo do projeto. Consiste em:
 1. Coletar requisitos.
 2. Definir o escopo.
 3. Criar a Estrutura Analítica de Processo (EAP).
 4. Verificar o escopo.
 5. Controlar o escopo.
3. Gerenciamento de tempo de projeto – descreve os processos necessários para garantir que a conclusão do projeto ocorra dentro do prazo previsto. Consiste em:
 1. Definir atividades.
 2. Sequenciar atividades.
 3. Estimar recursos da atividade.
 4. Estimar duração da atividade.
 5. Desenvolver o cronograma.
 6. Controlar o cronograma.
4. Gerenciamento de custos do projeto – detalha o gerenciamento dos custos envolvidos no projeto, de forma que os valores estabelecidos no planejamento sejam respeitados, executando-o dentro do orçamento estimado. No gerenciamento de custos, temos os processos para garantir que os custos para a execução do projeto não ultrapassem o que foi definido na etapa de planejamento. Consiste em:

1. Estimar de custos.
 2. Determinar o orçamento.
 3. Controlar custos.
5. Gerenciamento da qualidade do projeto – descreve os processos para garantir que o projeto vai atender às necessidades e expectativas do patrocinador ou usuário final. Consiste em criar um plano de gerenciamento para guiar você e sua equipe através das atividades relacionadas à qualidade. Consiste em:
1. Planejar a qualidade.
 2. Realizar a garantia da qualidade.
 3. Realizar o controle da qualidade.
6. Gerenciamento de recursos humanos do projeto – descreve os processos requeridos para otimizar as competências e habilidades em termos de alocação e realocação das pessoas envolvidas no projeto. Consiste em:
1. Desenvolver o plano de recursos humanos.
 2. Contratar ou mobilizar a equipe do projeto.
 3. Desenvolver a equipe de projeto.
 4. Gerenciar a equipe de projeto.
7. Gerenciamento das comunicações do projeto – descreve os passos requeridos para garantir o adequado uso, repasse, armazenamento e disseminação das informações do projeto. Consiste em:
1. Identificar as partes interessadas.
 2. Planejar as comunicações.
 3. Distribuir as informações.
 4. Gerenciar as expectativas das partes interessadas.
 5. Relatar desempenho.
8. Gerenciamento de riscos do projeto: descreve os processos relacionados na identificação, análise e respostas aos possíveis riscos que envolvem o projeto. Consiste em:
1. Planejar o gerenciamento de riscos.
 2. Identificar riscos.
 3. Realizar a análise qualitativa de riscos.
 4. Realizar a análise quantitativa de riscos.
 5. Planejar respostas aos riscos.
 6. Monitorar e controlar riscos.
9. Gerenciamento de aquisições do projeto – descreve os processos requeridos para aquisição de bens e serviços que não são supridos internamente pela organização e que são indispensáveis para o bom andamento do projeto. Consiste em:
1. Planejar aquisições.
 2. Conduzir aquisições.

3. Administrar aquisições.
 4. Encerrar aquisições.
10. Gerenciamento das partes interessadas: área de conhecimento adicionada a partir da 5^a edição do PMBOK que descreve os processos necessários para gerenciar as partes envolvidas no projeto, ou seja, as articulações de pessoas internas ou externas que impactam nas decisões do projeto. Consiste em:
1. Identificar as partes interessadas.
 2. Planejar o gerenciamento das mesmas.
 3. Gerenciar as partes.
 4. Controlar o envolvimento das partes.



O PMI oferece certificações para profissionais de gerenciamento de projetos e diversos padrões mundiais recomendados para as práticas de gestão de projetos. Maiores informações poderão ser extraídas do endereço: <<https://brasil.pmi.org/>>. Boa leitura!

9 GESTÃO DE PESSOAS EM PROJETOS DE SOFTWARE

Definir o escopo do projeto, determinar prazos e controlar os custos são algumas das muitas atividades do projeto, porém não há nada mais complexo no escopo de um projeto que gerenciar e tratar as expectativas de todos os envolvidos na sua execução, especialmente dos *stakeholders* do projeto.

O sucesso de um projeto é determinado pelas pessoas. Assim, quem, como e aonde vamos alocar alguém para desempenhar uma determinada função num projeto, pode comprometer todo seu desenvolvimento.

FONTE: Disponível em: <<http://eduacorrea.blogspot.com.br/2011/02/gestao-de-pessoas-nos-projetos-de.html>>. Acesso em: 9 set. 2015.

A última área de conhecimento do PMBOK adicionada a partir da sua 5^a edição é do gerenciamento das partes interessadas no projeto. O PMI descreve que esta área inclui os processos exigidos para identificar todos os “*Stakeholders*”, grupos ou organizações que podem impactar ou serem impactados pelo projeto. Uma equipe integrada de projeto exibe também características de trabalho em equipe, inclusive cooperação, comunicação aberta e alto nível de confiança entre os vários representantes que participam da equipe.

A gestão de pessoas também se concentra na comunicação contínua com as partes interessadas para entender suas necessidades e expectativas, abordando as questões conforme elas ocorrem, gerenciando os interesses conflitantes e incentivando o comprometimento das equipes com as decisões e atividades do projeto. A satisfação das pessoas deve ser gerenciada como um objetivo essencial do projeto.

FIGURA 36 – GESTÃO DE PESSOAS



FONTE: Disponível em: <http://www.perfiljr.com.br/wp-content/uploads/2012/12/gestao_de_pessoas_abril20121.jpg>. Acesso em: 18 ago. 2015.

Vargas (2009) cita que alocar a pessoa certa na atividade certa faz toda a diferença num setor onde cronogramas apertados ou irracionais são a principal fonte de riscos dos projetos desta natureza. Alocação das pessoas é um desafio que exige competências específicas do gerente de projeto no intuito de mapear as competências técnicas de cada colaborador para maior otimização da matriz de responsabilidades do cronograma.

Para garantir que os requisitos dos *Stakeholders* envolvidos no projeto estejam refletidos no planejamento, execução e operação de um projeto, a equipe do projeto deve ser multidisciplinar e representar todas as áreas de conhecimento necessárias.

Uma estrutura básica de uma equipe de desenvolvimento de *software* pode ser composta por:

- Gerente de projeto: coordenador do projeto.
- Analista de Negócios: especialista nas regras de negócio.

- Analista de Sistemas: faz a modelagem do sistema.
- Projetista de Sistemas: documenta as regras de negócio e desenha o aplicativo.
- Programadores: constroem os programas.
- Analista de qualidade: efetua os testes antes da liberação para o cliente.
- DBA: responsável pelo banco de dados.

De acordo com o PMBoK (2013), o gerenciamento da equipe do projeto é o processo de acompanhamento do desempenho dos membros da equipe, resolução de problemas, fornecimento de *feedback* e realização de mudanças de forma a melhorar o desempenho do projeto.

Nem sempre é possível conseguir as pessoas ideais para trabalharem num projeto devido a limitações do orçamento e não permitir o uso de pessoal altamente qualificado e, consequentemente, bem pago ou não estarem disponíveis pessoas com experiência apropriada a algum trabalho do projeto.

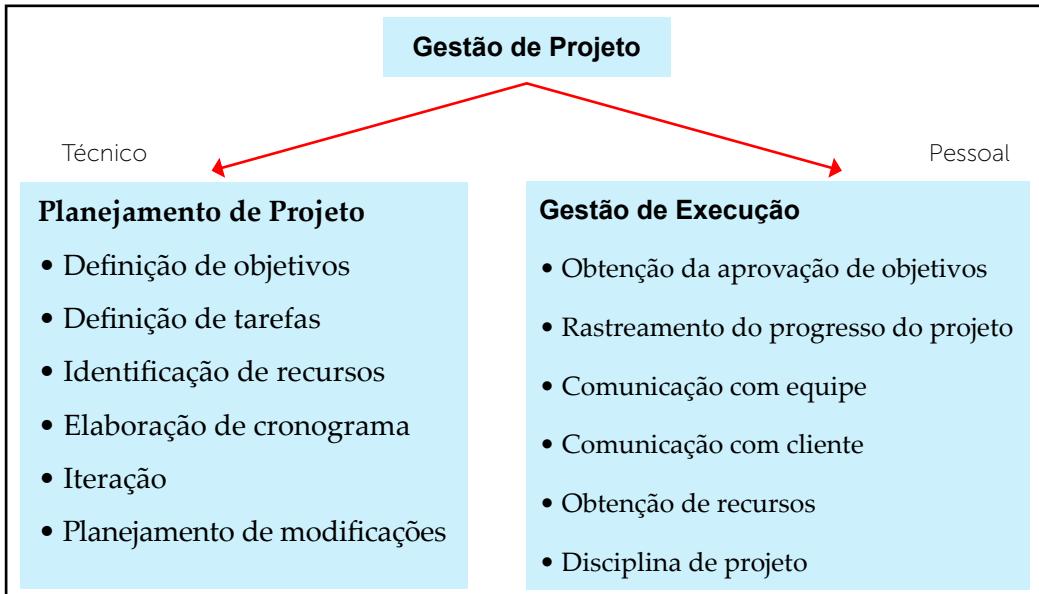
Pressman (2011, p. 569) diz que as características que definem um efetivo gerenciamento de projetos concentram-se em quatro aspectos-chave:

- Solução de problema: um gerente de projeto eficaz sabe diagnosticar itens técnicos e organizacionais que são os mais relevantes, sistematicamente estrutura uma solução ou motiva apropriadamente outros desenvolvedores a buscar a solução, põe em prática as lições aprendidas de projetos anteriores para novas situações e permanece suficientemente flexível para mudar de direção, caso as tentativas iniciais para a solução de problemas tenham sido infrutíferas.
- Identidade gerencial: um bom gerente de projeto deve assumir a responsabilidade do projeto. Deve ter confiança para assumir o controle quando necessário e deve assegurar que permitirá ao pessoal técnico seguir os seus instintos.
- Realizações: um gerente competente deve recompensar iniciativas e realizações para otimizar a produtividade de uma equipe. Deve demonstrar por meio de seus próprios atos que decisões por riscos controlados não serão punidas.
- Formação de equipes e de influência: um gerente efetivo deve ser capaz de "ler" pessoas, deve ser capaz de compreender sinais verbais e não verbais e reagir às necessidades das pessoas que estão enviando esses sinais. O gerente deve permanecer sobre controle em situações de alto estresse.

Em muitos casos, os problemas de gestão de projetos estão relacionados à natureza de comportamento. Motivação, satisfação, inovação, sentir-se importante, colaboração, experiência, conhecimento, disciplina, comunicação, respeito, autoestima, compromisso, desempenho, profissionalismo, frustração, criatividade e habilidades são exemplos de fatores intangíveis que regem as pessoas (MAGAZINE 55, 2015).

Segundo Filho (2015), três pilares formam a base da gestão de projetos: ter foco no cliente, fazer a equipe trabalhar bem (leia-se de forma produtiva e colaborativa) e administrar os recursos (de tempo, pessoal, financeiro) do projeto. A gestão de projetos de *software* pode ser vista sob duas perspectivas: técnica e pessoal onde a ênfase se dá sobre atividades de planejamento e execução, conforme ilustrado na figura.

FIGURA 37 – PERSPECTIVAS DA GESTÃO DE PROJETOS



FONTE: Disponível em: <<http://www.devmedia.com.br/gestao-de-projetos-de-software/9143>>. Acesso em: 16 ago. 2015.

Segundo Silva e César (2009), nos últimos anos, diversas pesquisas têm buscado aplicar teorias da psicologia à engenharia de *software* com o objetivo de obter teorias, técnicas e ferramentas específicas para projetos de *software* em dois aspectos complementares: na alocação de pessoas a papéis funcionais (técnicos e gerenciais) do desenvolvimento de *software*; e na composição e gerenciamento das equipes de desenvolvimento.

Por trás de todo projeto bem-sucedido está uma grande equipe. É responsabilidade do gerente de projetos manter a equipe concentrada, motivada e unida para atingir os objetivos do projeto. Essa é uma das áreas de conhecimento complicada para o GP, pois ele não pode confiar somente nos seus conhecimentos técnicos.

Para gerir de forma adequada o RH de um projeto, o GP deve possuir habilidades interpessoais e facilidade de relacionamento. Outras responsabilidades do GP no gerenciamento de RH incluem a manutenção de um bom ambiente de trabalho e a negociação de pessoas. Em organizações projetizadas, é comum organizar as pessoas em *pools* de acordo com suas habilidades. Por exemplo, uma empresa de desenvolvimento de *software* pode ter *pools* de analistas, *pools* de desenvolvedores, *pools* de designers etc. Caso algum recurso seja necessário, o GP pode ter de negociá-lo com o gerente de portfólio ou mesmo com outro GP.

Um importante componente do plano de RH é o Plano de Gerenciamento de Pessoal. Neste plano você identifica sua equipe e as necessidades de treinamento, estabelece os critérios para reconhecimento e recompensa pelo trabalho e ainda determina qual o período de trabalho de cada membro da equipe e a

disponibilidade ou não dos mesmos para trabalhar em outros projetos. Em projetos de TI normalmente é estabelecido que os recursos não trabalhem menos de um dia útil em cada projeto, devido ao fato de a produtividade ser prejudicada. Se um programador estiver alocado em dois projetos distintos, forçá-lo a trabalhar em um projeto pela manhã e outro projeto pela tarde vai diminuir consideravelmente sua produtividade, pois a mudança de contexto é bastante prejudicial.

10 FERRAMENTAS PARA GERÊNCIA DE PROJETOS

No desenvolvimento de um projeto existe a necessidade de um gerenciamento de projetos adequado, aplicando técnicas para o auxílio do controle das pessoas envolvidas e dos serviços atribuídos a elas, preocupando-se com os prazos, custos e benefícios de cada produto.

FONTE: Disponível em: <<http://www.devmedia.com.br/ferramentas-para-gerencia-de-projetos-engenharia-de-software-25/17147>>. Acesso em: 9 set. 2015.

Por isso, o uso de ferramentas para gerenciamento de projetos é indispensável, pois contribui para auxiliar e prover de forma rápida e eficiente as informações necessárias para o controle do trabalho realizado.

A adoção de ferramentas de gerenciamento de projetos pelas organizações efetivamente gera resultados positivos, propiciando padronização de métodos e processos de trabalho, além da disponibilização de informações em tempo real ao alcance de toda a equipe envolvida no projeto, aumentando a qualidade do gerenciamento e as chances de alcançar os objetivos traçados.

FONTE: Disponível em: <<http://www.devmedia.com.br/ferramentas-para-gerencia-de-projetos-engenharia-de-software-25/17147>>. Acesso em: 9 set. 2015.

Sendo assim, a seguir são destacadas quatro ferramentas para gerenciamento de projetos de *software* existentes no mercado:

10.1 MS PROJECT

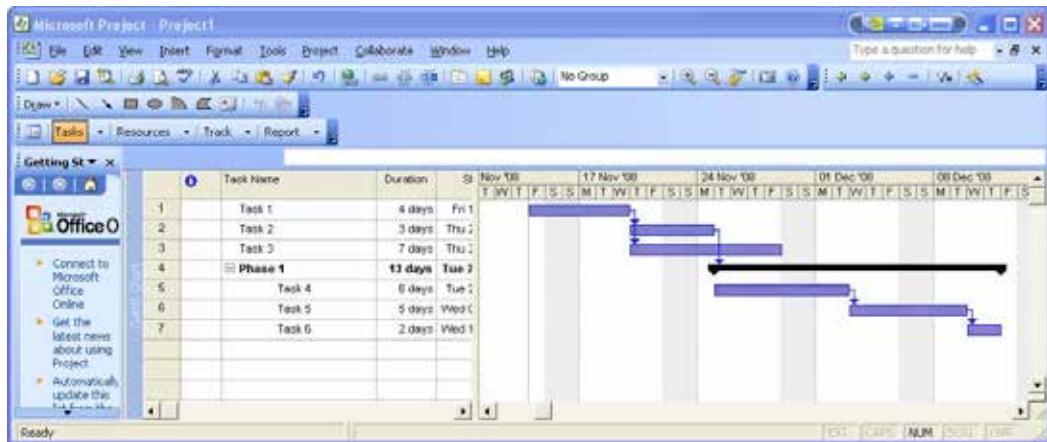
Um *software* de Gerenciamento de Projetos desenvolvido pela Microsoft Corporation, constituindo-se uma das ferramentas mais utilizadas pelo mercado em função dos recursos de gerenciamento que oferece e da facilidade operacional que apresenta.

FONTE: Disponível em: <<http://www.lem.ep.usp.br/Pef411/~Marcel%20Abe/Ms%20Project98.htm>>. Acesso em: 9 set. 2015.

A ferramenta de gerenciamento MS Project é aplicável aos mais diversos tipos de projetos, sejam na gestão de projetos de engenharia ou na área de gestão empresarial.

No geral, baseia-se no modelo Diagrama de Rede, utiliza tabelas no processo de entrada de dados, permite uso de subprojetos, possui recursos para agrupar, filtrar e classificar tarefas, possui um conjunto padrão de relatórios e os usuários podem criar seus próprios relatórios, permite definição do cronograma de trabalho.

FIGURA 38 – TELA DE UM PROJETO NO MS PROJECT



FONTE: Global Project Performance. Disponível em: <<http://www.globalprojectperformance.com/en/ms-project-epm/>>. Acesso em: 19 jun. 2015.

10.2 GANTT PROJECT

O Gantt Project é uma ferramenta de gerenciamento de projeto de acesso gratuito, de código aberto, baseado no gráfico de Gantt. O software é um sistema *desktop* que possui interface de fácil entendimento, com todos os seus recursos iniciais para começar a cadastrar as informações de um projeto.

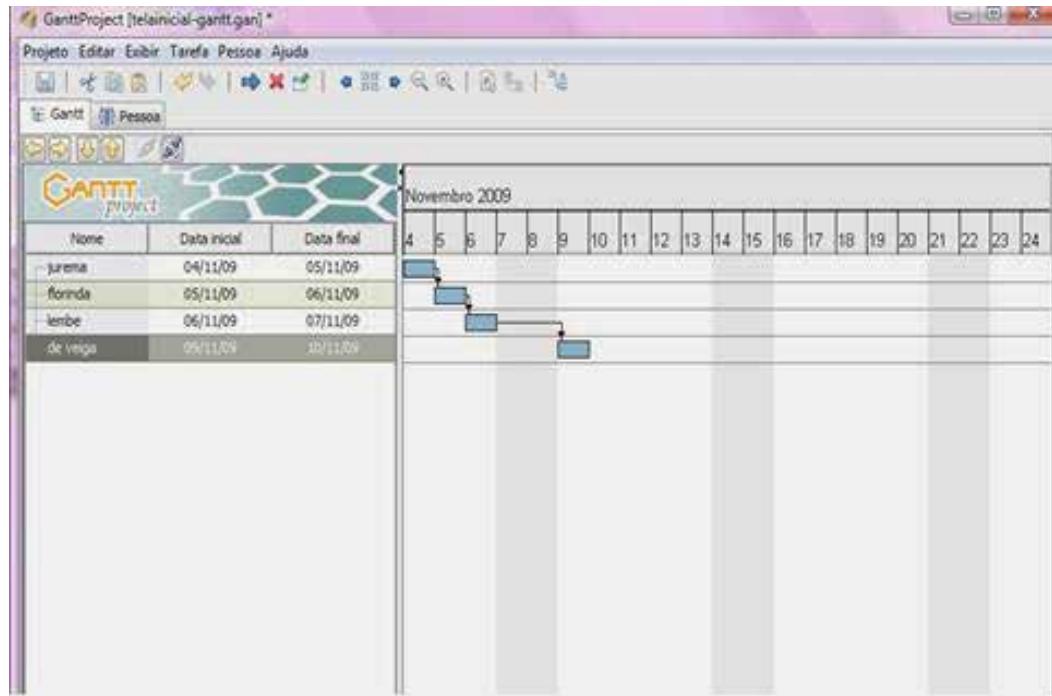
FONTE: Disponível em: <<http://www.devmedia.com.br/ferramentas-para-gerencia-de-projetos-engenharia-de-software-25/17147>>. Acesso em: 9 set. 2015.

Esta ferramenta possui o recurso de dividir o projeto em uma árvore de tarefas e atribuí-las ao responsável de cada uma, podendo criar dependências entre as tarefas, além da geração de relatórios nos formatos PDF e HTML, associação com aplicativos de planilha eletrônica, intercâmbio com o Microsoft Project, envio de *e-mail* para pessoas diretamente envolvidas no projeto, definição de calendários com feriados, e definição de novos atributos para as atividades (LEMME DE VEIGA, 2015).

O grande destaque dessa ferramenta é sua grande facilidade de uso e a clareza da interface. É recomendada para situações em que a definição e acompanhamento do cronograma é importante, assim como a necessidade de usá-lo em diversos sistemas operacionais e sua facilidade de uso.

FONTE: Disponível em: <<http://www.devmedia.com.br/ferramentas-para-gerencia-de-projetos-engenharia-de-software-25/17147>>. Acesso em: 9 set. 2015.

FIGURA 39 – TELA INICIAL DO GANTT PROJECT



FONTE: Gantt Project. Disponível em: <Ferramentas para Gerência de Projetos – Engenharia de Software 25 <http://www.devmedia.com.br/ferramentas-para-gerencia-de-projetos-engenharia-de-software-25/17147#ixzz3e6hLO6N6>>. Acesso em: 22 jun. 2015.

10.3 DOTPROJECT

O DotProject é uma ferramenta de gerência de projetos de *software* livre e código aberto. Tem como objetivo proporcionar ao gerente de projeto uma ferramenta para gerenciar e compartilhar as tarefas, horários, datas, *e-mail* e comunicação, dentre outras. Esta ferramenta é utilizada em várias aplicações e ambientes, desde pequenos escritórios, empresas, departamentos governamentais e escolas.

FONTE: Adaptado de: <<http://www.devmedia.com.br/ferramentas-para-gerencia-de-projetos-engenharia-de-software-25/17147>>. Acesso em: 9 set. 2015.

Tem como diferencial a sua operação via *web* e o uso de um banco de dados SQL, proporcionando bastante flexibilidade no uso dos dados.

O DotProject é recomendado para departamentos ou empresas em situações cujo o foco é a agenda de tarefas dos membros da equipe, gerenciamento da documentação associada aos projetos e apropriação de horas trabalhadas, com menos ênfase na manipulação de cronograma.

FONTE: Disponível em: <<http://www.devmedia.com.br/ferramentas-para-gerencia-de-projetos-engenharia-de-software-25/17147>>. Acesso em: 9 set. 2015.

FIGURA 40 – TELA DE UM PROJETO NO DOTPROJECT

The screenshot shows the DotProject software interface. At the top, there's a navigation bar with links for Companies, Projects, Tasks, Calendar, Files, Contacts, Forums, Tickets, User Admin, and System Admin. On the right side of the header, there are buttons for Help, My Info, Todo, Today, and Logout, along with a "New Item" dropdown. Below the header, a search bar and several action buttons (new task, new event, new file) are visible. The main content area is titled "View Project" and displays a project named "Prueba Vero 1". The left panel contains a "Details" section with project information: Company: ACME, Short Name: Prueba Ver, Start Date: 01/09/2008, Target End Date: 10/11/2008, Actual End Date: 25/09/2008, Target Budget: \$0.00, Project Owner: Person, Admin, URL:, and Staging URL:. The right panel contains a "Summary" section with status: In Progress, Priority: high, Type: Operative, Progress: 0.0%, Worked Hours: 24, Scheduled Hours: 24, and Project Hours: 24. Below these sections, there are tabs for Tasks, Tasks (Inactive), Forums, Gantt Chart, Task Logs, Events, and Files. A date range selector shows "From: 01/09/2008" and "To: 01/10/2008". There are also checkboxes for "Show captions", "Show work instead of duration", and "Sort by Task Name", with a "submit" button. Underneath the date range, there are links for "show this month" and "show full project". At the bottom of the main content area, there's a message: "We could not configure your memory or your memory configuration is too low, this may be as a result of Safe Mode being in effect and may result in not enough memory to display this chart." The bottom right corner of the interface shows the date "Today".

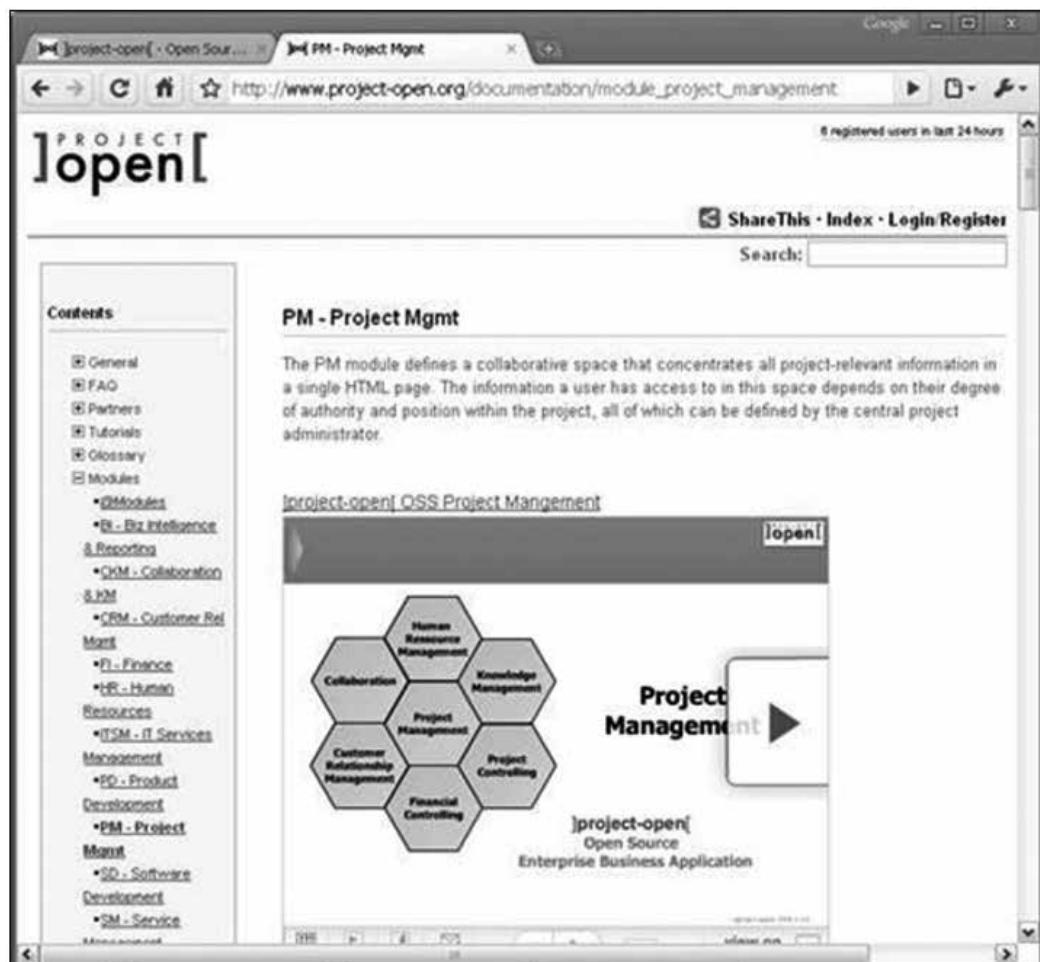
FONTE: Disponível em: <<http://www.tekchup.com/2008/09/ferramentas-para-gerenciar-projetos/>>. Acesso em: 22 jun. 2015

10.4 PROJECT OPEN

O Project Open é um aplicativo de código aberto baseado na *web*, e é usado por várias empresas que utilizam o sistema para gerenciar seus negócios. O *software* permite o monitoramento e o planejamento do projeto, permite a utilização de gráficos de Gantt através de interface com outras ferramentas, como a Gantt Project, calcula perdas e lucros por projetos e clientes, e define os orçamentos do projeto baseados em tempo ou custo absoluto (LEMBE DE VEIGA, 2015).

O *software* tem como principais objetivos a administração dos custos de um projeto e a colaboração entre os membros da equipe. Os projetos podem ser estruturados em qualquer nível de subprojetos e projetos-tarefas.

FIGURA 41 – TELA DO PROJECT OPEN



FONTE: Disponível em: <Ferramentas para Gerência de Projetos – Engenharia de Software 25 <<http://www.devmedia.com.br/ferramentas-para-gerencia-de-projetos-engenharia-de-software-25/17147#ixzz3e6hLO6N6>>. Acesso em: 22 jun. 2015

RESUMO DO TÓPICO 1

Neste tópico você aprendeu que:

- Projetos de *Software* são compostos por várias etapas exigindo mudanças rápidas, e para serem desenvolvidas com êxito, é exigido conhecimento em técnicas e ferramentas de gerenciamento/planejamento.
- O Instituto para Gerenciamento de Projetos (PMI) é uma entidade internacional sem fins lucrativos que reúne profissionais da área de gerenciamento de projetos de todo mundo e publica periodicamente diversos livros das melhores práticas e o mais conhecido é o PMBOK.
- O projeto segundo PMI é um conjunto de atividades temporárias com início e fim definidos no tempo, realizadas em grupo, destinadas a produzir um produto, serviço ou resultado único.
- Um projeto é considerado progressivo, pois à medida que avança nos conhecimentos, métodos e técnicas são geradas atendendo as particularidades e inovações.
- Gerência de projetos envolve planejamento, monitoração e controle de pessoas, processos e eventos que ocorrem à medida que o *software* evolui.
- Os principais problemas que os projetos enfrentam: atrasos nos prazos, estouro de orçamento, falta de qualidade nas entregas, falta de motivação da equipe e ocorrência de eventos inesperados ao longo da execução do projeto.
- Ricardo Viana Vargas (2009) enfatiza que o tripé de sucesso dos projetos é o tempo, custo e escopo, mas é incluído qualidade, pois não só basta entregar o produto dentro de um prazo e custo aceitável, mas precisa funcionar corretamente.
- O plano de projeto é um documento essencial, pois define os marcos de projeto e as principais atividades necessárias à sua execução.
- O gerenciamento de projeto no desenvolvimento de *software* tem um foco nos 4 Ps: pessoas (foco na comunicação e competência da equipe), produto (entender as necessidades e melhor solução), processo (fornecer metodologia e plano de projeto) e projeto (planejados e controlados com qualidade).
- Um gerente de projetos é peça-chave para que os planos sigam no rumo certo. O gerente de projetos quando está bem posicionado pode evitar riscos muito grandes no que está sendo desenvolvido.

- Os projetos podem ser compostos por três tipos de categorias: subprojetos, programas e portfólio. Subprojetos são diversos projetos menores. Programas são grupos de projetos relacionados e gerenciados coletivamente e portfólio é um conjunto de projetos ou programas com objetivos afins.
- A gestão de projetos é composta por cinco fases: conceitual, definição, produção, operacional e encerramento.
- As estruturas organizacionais são funcional, matricial e projetizada. Funcional quando as ordens vêm de cima para baixo, ou seja, decisões tomadas com auxílio dos gerentes de departamento. Projetada quando o GP é o único a tomar decisões ao projeto. As organizações matriciais ficam entre os dois extremos, dividindo-se em matriz fraca, balanceada ou forte.
- A análise e gestão de risco são ações que ajudam uma equipe de *software* a entender e gerenciar a incerteza, portanto, é aconselhável identificar e avaliar a probabilidade de ocorrência dos riscos, estimar seu impacto e estabelecer um plano de contingência caso o problema realmente ocorra.
- A gestão de riscos compreende resumidamente quatro etapas: (1) Identificação de riscos: busca identificar riscos de projeto ou de negócios. (2) Análise de risco: obter a probabilidade de ocorrência do risco e correspondente impacto sobre o projeto. (3) Administração de risco: estratégia de controle e elaboração do plano de contingência. (4) Monitoração de risco: detectar a ocorrência ou probabilidade de ocorrências.
- As quatro formas de se tratar um risco são: eliminação, mitigação, transferência e aceitação.
- O PMBOK (5^a edição) tem dez áreas de conhecimento que compõem no total 47 processos utilizando-se de entradas, ferramentas e técnicas e saídas para o gerenciamento de projetos, que são:
 1. Gerenciamento/Gestão de integração do projeto: descreve como as diversas áreas interagem entre si dentro de um projeto.
 2. Gerenciamento/Gestão do escopo do projeto: relacionado com os limites do produto ou serviço que será construído com o projeto.
 3. Gerenciamento/Gestão de tempo do projeto: garantir que a conclusão do projeto ocorra dentro do prazo previsto.
 4. Gerenciamento/Gestão de custos do projeto: fazer com que os valores estabelecidos no planejamento sejam respeitados, executando-o dentro do orçamento estimado.
 5. Gerenciamento/Gestão da qualidade do projeto: garantir que o projeto vai atender às necessidades e expectativas do patrocinador ou usuário final.
 6. Gerenciamento/Gestão de recursos humanos do projeto: otimizar as competências e habilidades em termos de alocação e realocação das pessoas envolvidas no projeto.

7. Gerenciamento/Gestão das comunicações do projeto: garantir o adequado uso, repasse, armazenamento e disseminação das informações do projeto.
 8. Gerenciamento/Gestão de riscos do projeto: relacionados na identificação, análise e respostas aos possíveis riscos que envolvem o projeto.
 9. Gerenciamento/Gestão de aquisições do projeto: descreve os processos requeridos para aquisição de bens e serviços que não são supridos internamente pela organização.
 10. Gerenciamento/Gestão de envolvidos do projeto: gerenciar as articulações de pessoas internas ou externas que impactam nas decisões do projeto.
- O sucesso de um projeto é determinado pelas pessoas, pois a satisfação das pessoas deve ser gerenciada como um objetivo essencial do projeto.
 - Não há nada mais complexo no escopo de um projeto que gerenciar e tratar as expectativas de todos os envolvidos na sua execução, especialmente dos *Stakeholders* do projeto.
 - Uma estrutura básica de uma equipe de desenvolvimento de *software* pode ser composta por Gerente de projeto, Analista de Negócios, Analista de Sistemas, Projetista de Sistemas, Programadores, Analista de qualidade e DBA.
 - Filho (2015) apresentou três pilares que formam a base da gestão de projetos: (1) ter foco no cliente, (2) fazer a equipe trabalhar bem e (3) administrar os recursos (de tempo, pessoal, financeiro) do projeto. Portanto, ela pode ser vista sob duas perspectivas: técnica e pessoal onde a ênfase se dá sobre atividades de planejamento e execução.
 - O uso de ferramentas para gerenciamento de projetos é indispensável, para auxiliar o GP existem as ferramentas, desde as mais simples às mais elaboradas, e foram citadas neste tópico: MS Project, Gantt Project, DotProject e Project Open utilizadas para auxiliar e prover de forma rápida e eficiente as informações necessárias para o controle do trabalho realizado nos projetos.

AUTOATIVIDADE



- 1 As organizações estão progressivamente empreendendo projetos que sejam um tanto incomuns ou únicos, tenham prazos finais específicos, contenham tarefas complexas inter-relacionadas que exijam habilidades especializadas e sejam de caráter:
 - a) () rotineiro.
 - b) () contínuo.
 - c) () duráveis.
 - d) () temporário.
 - e) () operacional.
- 2 Qual dos fatores a seguir não está incluído na “Restrição Tripla”?
 - a) () Escopo.
 - b) () Tempo.
 - c) () Custo.
 - d) () Aparência.
 - e) () Qualidade.
- 3 De acordo com o Guia PMBOK, os processos necessários para assegurar que o projeto inclua todo o trabalho necessário, e apenas o necessário, para que o projeto termine com êxito pertencem à seguinte área de conhecimento:
 - a) () Gerenciamento do Escopo do Projeto.
 - b) () Gerenciamento do Tempo do Projeto.
 - c) () Gerenciamento dos Custos do Projeto.
 - d) () Gerenciamento das Comunicações do Projeto.
 - e) () Gerenciamento dos Recursos Humanos do Projeto.
- 4 Pressman (2011) define que o gerenciamento de projeto no desenvolvimento de *software* tem um foco nos 4 Ps, os quais iremos abordar a seguir.
 - I. Pessoal
 - II. Produto
 - III. Processo
 - IV. Projeto

() São planejados e controlados minuciosamente, pois esta é a maneira de administrar sua complexidade e garantir qualidade.

() Antes mesmo de começar o projeto, deve-se estabelecer os objetivos do produto e considerar as soluções.

- () Este recurso foca na formação da equipe, comunicação, ambiente de trabalho, desenvolvimento de carreira, análise da competência e cultura de equipe.
- () Fornece a metodologia por meio da qual um plano de projeto abrangente para o desenvolvimento de *software* pode ser estabelecido.

De acordo com as sentenças acima, associe a sequência correta das definições dadas para cada um dos 4 Ps:

- a) () I – II – III – IV.
- b) () IV – II – I – III.
- c) () IV – I – II – III.
- d) () II – I – IV – III.

5 Os projetos podem ser compostos por três tipos de categorias: subprojetos, programas e portfólio. Segundo suas características, assinale abaixo com V para verdadeiro ou F para falso:

- () Projetos grandes e complexos podem ser divididos em projetos menores chamados subprojetos.
- () Portfólio é um grupo de projetos que são relacionados e gerenciados coletivamente de forma coordenada.
- () Programa é composto por um conjunto de projetos ou subprojetos com objetivos comuns.
- () Programas podem não ser necessariamente interdependentes ou diretamente relacionados.
- () Um projeto pode ou não fazer parte de um programa, mas um programa sempre terá projetos.

6 O PMBOK (2015) classifica as organizações em diferentes estruturas de acordo com o grau de aderência às práticas sugeridas para a gerência de projetos, conforme listados abaixo:

- I. Funcional
- II. Matricial
- III. Projetizada

- () Somente o Gerente de Projetos toma as decisões referentes ao projeto, realiza os controles sobre os recursos e o orçamento do projeto.
- () As organizações ficam entre os dois extremos, dividindo-se em matriz fraca, balanceada ou forte.
- () Onde praticamente todas as decisões de gerenciamento de projetos são tomadas com auxílio dos gerentes de departamento.

De acordo com as sentenças acima, associe a sequência correta das definições apresentadas:

- a) () I – II – III.
- b) () III – II – I.
- c) () III – I – II.
- d) () II – I – III.

7 Sobre gerenciamento de riscos de *software*, assinale a alternativa CORRETA:

- a) () O gerenciamento de riscos de *software* não envolve os riscos ambientais, que possam afetar o projeto.
- b) () O controle de riscos é feito pelos planos de casos de uso.
- c) () O controle de riscos é realizado por membros externos ao projeto.
- d) () Gerenciamento de riscos de *software* consiste em avaliar e controlar os riscos, que afetam o projeto, processo ou produto de *software*.
- e) () O gerenciamento de riscos de *software* consiste, apenas, no gerenciamento dos Testes de *Software*.

8 Quais são as áreas de conhecimento definidas pela 5^a edição do guia PMBOK para o gerenciamento dos projetos?

9 Dadas as definições sobre as principais áreas de gerenciamento de projeto (5^a edição do guia PMBOK), associe os itens com suas respectivas áreas:

- I. Descreve os processos necessários para garantir que a conclusão do projeto ocorra dentro do prazo previsto.
- II. Descreve os processos para garantir que o projeto vai atender às necessidades e expectativas do patrocinador ou usuário final.
- III. Descreve os passos requeridos para garantir o adequado uso, repasse, armazenamento e disseminação das informações do projeto.
- IV. Descreve os processos relacionados na identificação, análise e respostas aos possíveis riscos que envolvem o projeto.

A ordem correta das áreas de conhecimento no gerenciamento definidas acima é:

- a) () Gerenciamento de tempo de projeto; Gerenciamento da qualidade do projeto; Gerenciamento das comunicações do projeto; e Gerenciamento de riscos do projeto.
- b) () Gerenciamento da qualidade de projeto; Gerenciamento de riscos do projeto; Gerenciamento da integração do projeto; e Gerenciamento de escopo do projeto.
- c) () Gerenciamento de escopo de projeto; Gerenciamento da qualidade do projeto; Gerenciamento de RH do projeto; e Gerenciamento de aquisição do projeto.

- d) () Gerenciamento de tempo de projeto; Gerenciamento de riscos do projeto; Gerenciamento de custos do projeto; e Gerenciamento das partes interessadas do projeto.

10 Dadas as definições abaixo sobre as principais áreas de gerenciamento de projeto (5^a edição do guia PMBOK), associe os itens com suas respectivas áreas:

- I. Está relacionado com os limites do produto ou serviço que será construído com o projeto.
- II. Detalha o gerenciamento dos custos envolvidos no projeto, de forma que os valores estabelecidos no planejamento sejam respeitados, executando-o dentro do orçamento estimado.
- III. Descreve os processos requeridos para otimizar as competências e habilidades em termos de alocação e realocação das pessoas envolvidas no projeto.
- IV. Descreve os processos necessários para garantir que a conclusão do projeto ocorra dentro do prazo previsto.

A ordem correta das áreas de conhecimento no gerenciamento definidas acima é:

- a) () Gerenciamento de qualidade do projeto; Gerenciamento de aquisição do projeto; Gerenciamento de comunicação do projeto; e Gerenciamento de custos de projeto.
- b) () Gerenciamento do escopo do projeto; Gerenciamento de tempo do projeto; Gerenciamento de comunicação do projeto; e Gerenciamento de integração de projeto.
- c) () Gerenciamento do requisito do projeto; Gerenciamento de aquisição do projeto; Gerenciamento de RH do projeto; e Gerenciamento de custos de projeto.
- d) () Gerenciamento do escopo do projeto; Gerenciamento de custos do projeto; Gerenciamento de RH do projeto; e Gerenciamento de tempo de projeto.

11 Acerca das definições dadas sobre gestão de pessoas em projetos de *software*, assinale V para verdadeiro ou F para falso:

- () O sucesso de um projeto depende exclusivamente dos seus processos e ferramentas adotadas ao invés da definição de como e quem desempenhará a função do projeto.
- () Uma equipe integrada de projeto precisa trabalhar colaborativamente utilizando uma comunicação aberta, porém o nível de confiança entre os vários representantes que participam da equipe se faz desnecessário para o sucesso do projeto.

- () Para garantir que os requisitos dos *Stakeholders* envolvidos no projeto estejam refletidos no planejamento, execução e operação de um projeto, a equipe do projeto deve ser multidisciplinar e representar todas as áreas de conhecimento necessárias.
- () O gerenciamento da equipe do projeto é o processo de acompanhamento do desempenho dos membros da equipe, resolução de problemas, fornecimento de *feedback* e realização de mudanças de forma a melhorar o desempenho do projeto.
- () A gestão de projetos de *software* pode ser vista sob duas perspectivas: técnica e pessoal, onde a ênfase se dá sobre atividades de planejamento e execução.

ESTIMATIVAS E MÉTRICAS DE PROJETOS DE SOFTWARE

1 INTRODUÇÃO

A tarefa de gerenciar projeto de *software* engloba, além do minucioso processo de planejamento e definição dos trabalhos e de seu gerenciamento, a definição e escolha de bons orçamentos que tragam valor agregado ao processo, e ainda, o controle de tais recursos de forma a cumprir com aquilo que foi definido de inicial. Estimar *software* significa determinar quanto de dinheiro, esforço, recursos e tempo serão necessários para criar um sistema, ou seja, o gerente e a equipe de desenvolvimento devem estimar o trabalho a ser realizado, os recursos necessários, o tempo de duração e, por fim, o custo do projeto.

A estimativa de custo faz parte do planejamento de qualquer projeto de engenharia. A diferença é que na engenharia de *software* o custo principal é o esforço, ou seja, o custo de mão-de-obra; assim, para se calcular o custo de *software*, é necessário dimensionar o trabalho para desenvolvê-lo. Em geral, esse trabalho é expresso pelo número de pessoas trabalhando numa unidade de tempo, tal como pessoas-mês ou pessoa-ano. O trabalho que uma pessoa consegue fazer num mês pode ser traduzido em números de horas de trabalho num mês.

Um projeto que envolva o desenvolvimento de *software* inclui as dificuldades em se manter os custos iniciais, baseados nos requisitos levantados no início do projeto até o término dele. Nisso está a importância, associada a estimativas de custos, também do escopo e tempo em questão.

Primeiro é feita a avaliação do escopo para saber a complexidade do trabalho a ser realizado e a partir do perfil dos profissionais que irão trabalhar no projeto irá determinar o tempo de execução do projeto. Quanto mais experiência e domínio tiverem maior produtividade e qualidade o projeto irá ter, porém, para isso ocorrer o valor hora dos profissionais serão bem diferentes dependendo do seu *know-how* no assunto. Normalmente, as empresas utilizam o valor de 187 horas mês de trabalho de cada funcionário como parâmetro de cálculo do esforço do projeto. Assim um projeto com 10 pessoas-mês, tem que trabalhar equivalente a 1870 horas, que deverão ser distribuídas no prazo estimado do projeto.

Por fim, pode-se afirmar que o esforço está relacionado à produtividade, que é medida pela qualidade de trabalho realizada por uma unidade de esforço.

Uma forma de medida de produtividade é, por exemplo, a quantidade de linha de código pessoa-mês.

2 O GERENCIAMENTO DE CUSTOS: GUIA PMBOK

De acordo com o Guia PMBOK 5^a edição, escrito pelo *Project Management Institute – PMI* (2013), o gerenciamento de custos compreende quatro processos principais, são eles:

- Planejar Gerenciamento dos Custos: descreve como serão realizados os processos de estimar custos, definir orçamento e controlar fluxo de caixa, definindo as ferramentas e técnicas a serem utilizadas, padrões e políticas a serem observadas;
- Estimar Custos: compreende a estimativa dos custos dos componentes do projeto;
- Definir Orçamento: envolve a previsão dos custos agregados, simulações e adequação às restrições; e,
- Controlar Custos: processo responsável pelo acompanhamento da execução orçamentária, utilizando indicadores e envolvendo a tomada de ações corretivas ou preventivas quando necessário.

As tarefas de estimar custos e controlá-los são as que demandam maior esforço do gerente e para a execução dos processos referentes ao gerenciamento de custos, três itens são importantes: as entradas, as ferramentas e técnicas e as saídas.

As entradas se referem às informações ou dados obtidos referentes ao projeto, oriundos de fatores internos ou externos (pessoas, instalações, equipamentos, materiais etc.). Como entrada do gerenciamento de custos podemos citar o calendário da utilização de equipamentos alugados durante o projeto, os planos de riscos contendo quais atividades tem com maior impacto financeiro no projeto, entre outros.

Já as ferramentas e técnicas utilizadas podem ser um padrão que formaliza os procedimentos a serem adotados para efetiva administração dos custos do projeto. Como exemplo se tem diversos custos relacionados à qualidade, treinamentos, auditorias, retestes, replanejamento de projetos ou mesmo o valor de licença para utilização de alguma ferramenta CASE da engenharia de software para o gerenciamento de projetos, para o gerenciamento de configuração, testes automatizados, entre outros.

Por sua vez, as saídas são produtos, como as estimativas de custos, recursos, cronogramas, tempo das atividades, previsões orçamentárias, entre outros. As

saídas são elementos que também podem variar muito, desde atualizações no plano de gerenciamento de projeto, passando por medidas de *performance* de trabalho, indo até uma *baseline* de custos e necessidades de financiamento do projeto.

3 MÉTRICAS DE SOFTWARE

Métricas de *Software* é um assunto discutido há mais de vinte anos na engenharia de *software*, e, no entanto, não é verificada sua utilização, na prática, pela grande maioria dos projetos de construção de *software*. Área que possibilita realizar uma das atividades mais fundamentais do processo de gerenciamento de projetos: o planejamento. A partir desse, pode-se identificar a quantidade de esforço, de custo e das atividades que serão necessárias para a realização do projeto.

Uma métrica *software* é a medição de um atributo (propriedades ou características) de uma determinada entidade (produto, processo ou recursos). Como exemplo podemos citar: o tamanho do produto de *software* em número de linhas de código; número de pessoas necessárias para programar um caso de uso; número de defeitos encontrados por fase de desenvolvimento; esforço para a realização de uma tarefa; tempo para a realização de uma tarefa; custo para a realização de uma tarefa; grau de satisfação do cliente etc.

A partir do uso das métricas de *software* uma empresa desenvolvedora de sistemas poderá entender e aperfeiçoar o processo de desenvolvimento, melhorar a gerência de projetos e o relacionamento com clientes, reduzir frustrações e pressões de cronograma, gerenciar contratos de *software*, indicar a qualidade de um produto de *software*, avaliar a produtividade do processo, avaliar os benefícios (em termos de produtividade e qualidade) de novos métodos e ferramentas de engenharia de *software*, avaliar retorno de investimento, identificar as melhores práticas de desenvolvimento de *software*, embasar solicitações de novas ferramentas e treinamento, avaliar o impacto da variação de um ou mais atributos do produto ou do processo na qualidade e/ou produtividade, formar uma *baseline* para estimativas, melhorar a exatidão das estimativas, oferecer dados qualitativos e quantitativos ao gerenciamento de desenvolvimento de *software*, de forma a realizar melhorias em todo o processo de desenvolvimento de *software* etc.

Há pouco tempo, a única base para a realização de estimativas era a experiência da equipe técnica envolvida no projeto, um processo subjetivo e que eventualmente levava a atividades atropeladas ou não realizadas, produtos com deficiência funcional, custo de realização além do previsto e atraso na entrega do produto. Um dos grandes problemas da utilização da experiência passada de desenvolvimento de projetos de *software* em novos desenvolvimentos é a dificuldade de estabelecer semelhanças de funcionalidades e tamanho entre projetos de *software*.

FONTE: Adaptado de: <<http://www.batebyte.pr.gov.br/modules/conteudo/conteudo.php?conteudo=88>>. Acesso em: 9 set. 2015.

Segundo Sommerville (2003), uma métrica de *software* é qualquer tipo de medição que se refira a um sistema de *software*, processo ou documentação relacionada. Essa métrica tem como principal objetivo especificar as funções de coleta de dados de avaliação e desempenho, atribuindo essas responsabilidades a toda a equipe envolvida no projeto e analisando os históricos dos projetos anteriores. Coletando dados para essas medições, as dúvidas em relação ao *software* poderão ser respondidas podendo realizar confirmações de que as melhorias do *software*, alcançaram ou não, as metas propostas.

4 MÉTRICAS UTILIZADAS PARA ESTIMAR SISTEMAS

De acordo com o livro de introdução à engenharia de *Software* escrito por Carvalho (2001), existem vários métodos que podem ser utilizados para se estimar o custo do desenvolvimento e a vida útil de um sistema. Em geral representa o custo monetário ou o esforço necessário para desenvolver e manter um sistema.

Para se estabelecer essas estimativas, pode-se utilizar técnicas de decomposição do produto e utilizar a opinião de especialistas que, baseados em experiências de projetos anteriores, são capazes de estimar o esforço e o tempo de desenvolvimento do projeto. Podem-se considerar dois tipos de decomposição: (1) decomposição do produto para estimar o número de linhas de código utilizando técnicas como pontos por função, caso de uso etc.; (2) decomposição do processo considerando-se as atividades de cada etapa de engenharia de *software*, dependente do paradigma utilizado.

A seguir são apresentados métodos ou técnicas para estimativa de *software*, que são: Linhas de Código (LOC); Pontos de História; Análise de Pontos de Função, Análise de Pontos de Caso de Uso, COCOMO II e estimativa para projetos Orientados a Objeto.

4.1 LINHA DE CÓDIGO (LOC)

É uma métrica orientada ao tamanho do *software*. A técnica conhecida de LOC (*Lines of Code*) ou SLOC (*Source Lines of Code*) foi possivelmente a primeira a surgir e consiste em estimar o número de linhas que um programa deverá ter, normalmente a partir da opinião de especialistas e histórico de projetos passados. Hoje esta técnica evoluiu para KSLOC (*Kilo Source Lines of Code*), tendo em vista que o tamanho da maioria dos programas passou a ser medido em milhares de linhas (OLIVEIRA FILHO, 2013).

Esta técnica começa quando se reunirá a equipe ou os principais representantes do projeto para discutir o sistema a ser desenvolvido, onde cada participante dará a sua opinião sobre a quantidade de KSLOC que serão necessários para desenvolver o sistema.

Usualmente a reunião não chegará a um valor único e deverão então ser considerados três valores:

- O KSLOC otimista, ou seja, o número mínimo de linhas que se espera desenvolver se todas as condições foram favoráveis.
- O KSLOC pessimista, ou seja, o número máximo de linhas que se espera desenvolver ante condições desfavoráveis.
- O KSLOC esperado, ou seja, o número de linhas que efetivamente se espera desenvolver em uma situação de normalidade.

A partir dos dados coletados, seus valores são aplicados na fórmula:

$$\text{KSLOC} = 4 * (\text{KSLOC esperado} + \text{KSLOC otimista} + \text{KSLOC pessimista}) / 6.$$



Este método é o mais simples, direto e altamente utilizado, porém, o uso apresenta algumas desvantagens. Primeiro, verifica-se que ela é fortemente ligada à tecnologia empregada, sobretudo a linguagem de programação. Segundo, pode ser difícil estimar essa grandeza no início do desenvolvimento, sobretudo se não houver dados históricos relacionados com a linguagem de programação utilizada no projeto.

4.2 PONTOS DE HISTÓRIA

É uma métrica de estimativa de tempo, onde é a estimativa de esforço preferida (embora não exclusiva) de métodos ágeis como Scrum e XP. Um ponto de história, não é uma medida de complexidade funcional como pontos de função ou pontos de caso de uso, mas uma medida de esforço relativa à equipe de desenvolvimento (SCHOFIELD; ARMEMTROUT; TRUJILLO, 2013).

Uma estimativa baseada em pontos de histórias deve ser feita pela equipe. Inicialmente pergunta-se à equipe quanto tempo tantas pessoas que se dedicassem a uma história de usuário levaria para terminá-la, gerando uma versão executável funcional. Se a resposta for, por exemplo, "5 pessoas levariam 3 dias", então atribua à história $5 \times 3 = 15$ pontos de história.

Portanto, um ponto de história pode ser definido como esforço de desenvolvimento de uma pessoa durante um dia ideal de trabalho, lembrando que o dia ideal de trabalho consiste em uma pessoa dedicada durante 6 a 8 horas a um projeto, sem interrupções nem atividades paralelas.

Nos métodos ágeis, a importância da estimativa normalmente está na comparação entre histórias, ou seja, mais importante do que saber quantos dias uma história efetivamente levaria para ser implementada é saber que uma história levaria duas vezes mais tempo do que outra para ser implementada.

4.3 ANÁLISE DE PONTOS DE FUNÇÃO (APF)

A *Function Point Analysis* (FPA) que significa Análise de Pontos de Função (APF) tem por objetivo definir processos e técnicas formais e padronizadas para dimensionamento e estimativa da complexidade de sistema, ou seja, para medir o tamanho do escopo do projeto (IFPUG, 2015).



A APF foi proposta por Allan Albrecht em 1979 e foram formalizadas suas regras de contagem a partir de 1984 pela IBM, e em 1986 foi criado o IFPUG (*International Function Points Users Group* - <www.ifpug.org>) orgão responsável por criar seu manual de contagem que encontra-se na versão 4.3.1. No Brasil é o *Brazilian Function Point Users Group* - <www.bfpug.com.br>.

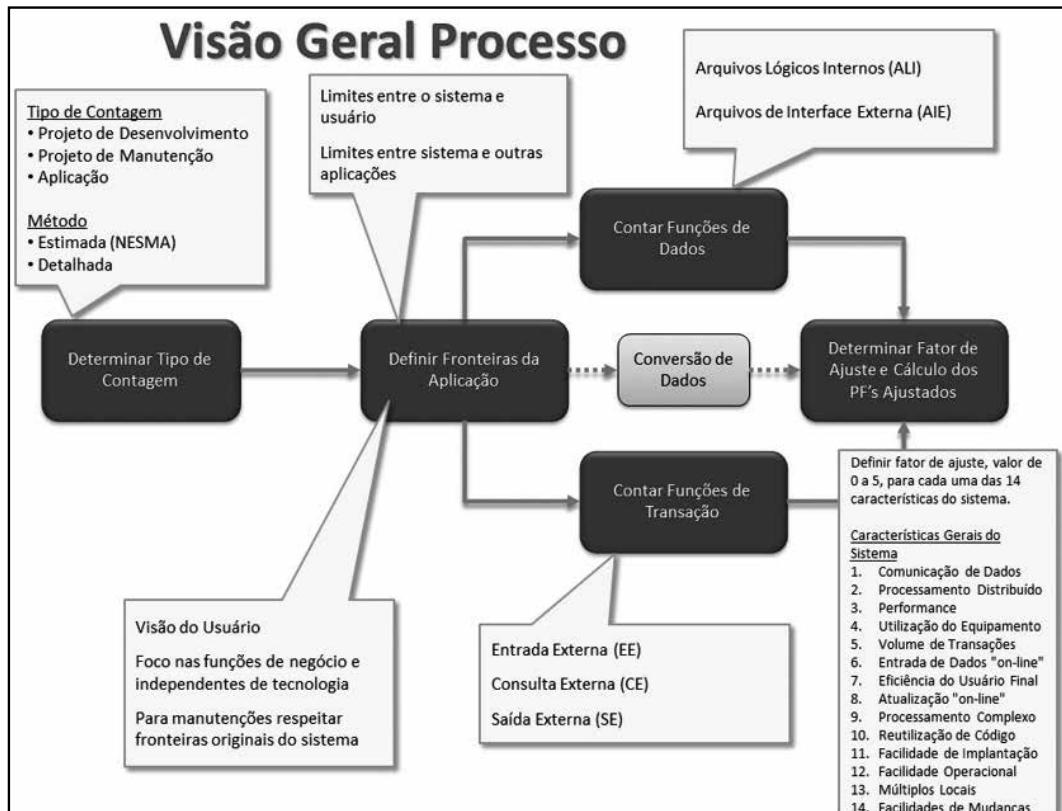
Neste método APF é possível quantificar as funções de um sistema considerando aspectos significantes para o usuário, portanto, irá considerar os requisitos de negócio que o sistema atende e principalmente que sua contagem é independente de tecnologia e, por isso, não é influenciada pela mesma e independe de plataforma ou linguagem de programação e estilo de programação utilizado (SCHOFIELD; ARMEMTROUT; TRUJILLO, 2013).

Segundo Reinaldo (2009), o IFPUG juntamente com o grupo que estuda APF na Europa, a Associação Holandesa de Métricas (NESMA), permite estimar o tamanho do sistema nas fases iniciais do projeto, em que as funcionalidades ainda não estão bem definidas. Porém, para manter e determinar os procedimentos de contagem APF definiu os seguintes passos:

- Determinar o tipo de contagem (desenvolvimento, melhoria ou aplicação existente).
- Determinar as Fronteiras da aplicação (escopo do sistema).
- Identificar e atribuir valor em pontos de função não ajustados para as transações sobre dados (entrada, consultas e saídas externas).
- Identificar e atribuir valor em pontos de função não ajustado (UFPA) para os dados estáticos (arquivos internos e externos).
- Determinar o valor de ajuste técnico (VAF).
- Calcular o número de pontos de função ajustados (AFP).

Estes passos poderão seguir um padrão simplificado de contagem conforme a figura abaixo, que mostra os procedimentos de contagem simplificados.

FIGURA 42 – PROCEDIMENTO DE CONTAGEM (SIMPLIFICADO)



FONTE: Adaptado pelo autor

Reinaldo (2009) informa que para determinar o tipo de contagem, deve avaliar primeiro as características do projeto se é um projeto de novo desenvolvimento ou manutenção e depois disso realiza-se através das definições vindas pela proposta do NESMA a escolha do método de contagem que são:

a) Estimado

- Conceitos da Análise de Pontos de Função (APF) do IFPUG (*International Function Point Users Group*) e calcula os pontos de função conforme orientação da NESMA (*Netherlands Software Metrics Association*).
- Funções de dados complexidade baixa e funções de transação complexidade média.

b) Detalhado

- A contagem detalhada utiliza os conceitos da Análise de Pontos de Função do Manual de Práticas de Contagem - versão 4.3.1 do IFPUG (CPM – *Counting Practices Manual*).
- Complexidade de funções de dados e de transação dependente dos elementos envolvidos em cada caso.

A técnica FPA avalia as duas naturezas dos dados. Os dados estáticos são a representação estruturada dos dados, na forma de arquivos internos ou externos e os dados dinâmicos são a representação das transações sobre os dados na forma de entradas, saídas e consultas externas. Abaixo se apresenta seus tipos (IFPUG, 2015):

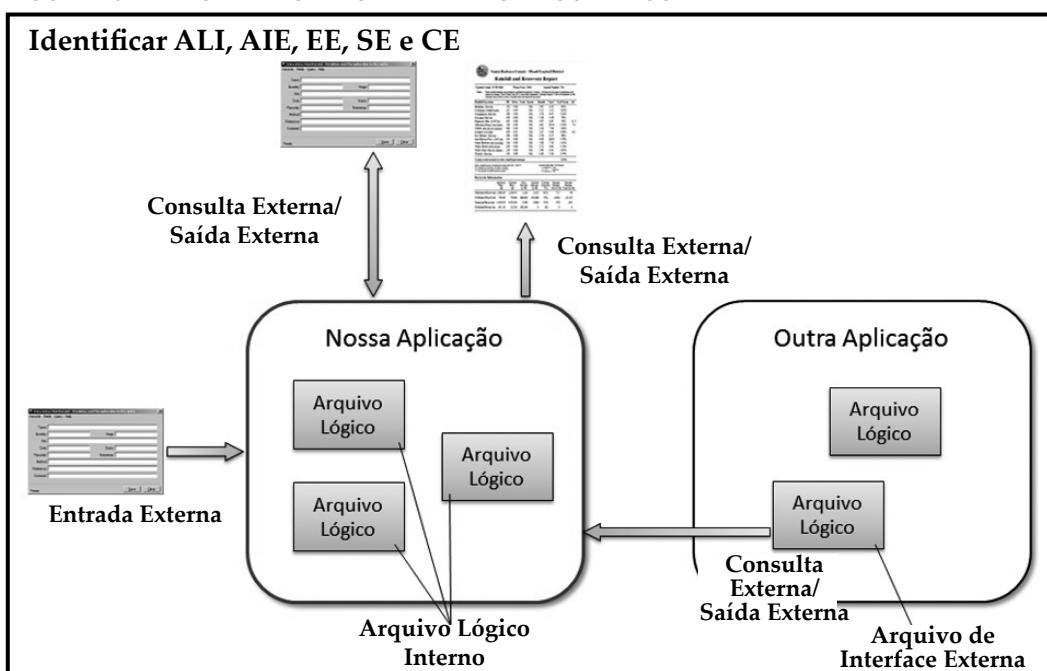
a) Tipos de funções estáticas

- Arquivo lógico interno (ALI): é um elemento do modelo conceitual percebido pelo usuário e mantido internamente pelo sistema. Consiste no grupo lógico de dados ou informações de controle, relacionados e reconhecidos pelo usuário.
- Arquivo de interface externa (AIE): é um elemento do modelo conceitual percebido pelo usuário e mantido externamente por outras aplicações. Consiste no grupo de dados, ou informações de controle, é lógico e identificável pelo usuário. Referenciado pela aplicação e mantido dentro da fronteira de outra aplicação.

b) Tipos de funções dinâmicas

- Entradas externas (EE): são entradas de dados ou controle, que têm como consequência a alteração do estado interno das informações do sistema.
- Saídas externas (SE): são saídas de dados que podem ser precedidas ou não da entrada de parâmetros. Pelo menos um dos dados de saída deve ser derivado, ou seja, calculado.
- Consultas externas (CE): são saídas de dados que podem ser percebidas ou não da entrada de parâmetros. Os dados devem sair da mesma forma como estavam armazenados, sem transformações ou cálculos.

FIGURA 43 – NATUREZA ESTÁTICA E DINÂMICA DOS DADOS



FONTE: O autor (adaptado)

Parâmetros para estimar complexidade de funções (IFPUG, 2015):

- Registro RET (*Record Element Type*) que corresponde a um subconjunto de dados reconhecível pelo usuário dentro de um arquivo interno ou externo (uma classe qualquer).
- Arquivo FTR (*File Types Referenced*) que corresponde a um arquivo interno ou externo, usado em uma transação (uma classe que não seja componente de outra).
- Argumento DET (*Data Element Type*) que corresponde a uma unidade de informação (um campo), a princípio indivisível e reconhecível pelo usuário, normalmente seria um campo de uma tabela, um atributo de uma classe ou um parâmetro de uma função.

A contagem APF é feita através de uma tabela definida pelo IFPUG (figura abaixo) onde para cada função de dados são contados os tipos de registros (RET) e a quantidade de campos de dados que ele possui e para cada função de transação são contados os arquivos referenciados (FTR) e a quantidade de campos de dados destes arquivos.

FIGURA 44 – CONTAGEM DOS ELEMENTOS

ALI e AIE				SE e CE				EE				
Tipos de Registro (TR)	Tipo de Dados (TD)			Arquivos Referenciados (AR)	Tipo de Dados (TD)			Arquivos Referenciados (AR)	Tipo de Dados (TD)			
	<20	20-50	>50		<6	6-19	>19		<2*	Baixa	Baixa	Média
	1	Baixa	Baixa		Média	2-3	Baixa		Média	>3	Média	Alta
2-5	Baixa	Média	Alta	>2*	Baixa	Média	2-3	Baixa	Média	Alta		
>5	Média	Alta	Alta	>3	Média	Alta	>2	Baixa	Baixa	Média		

OU											
Argumentos DET			Argumentos DET			Argumentos DET					
Classes RET	1 a 19	20 a 50	51 ou mais	Classes FTR	1 a 5	6 a 19	20 ou mais	Classes FTR	1 a 4	5 a 15	16 ou mais
1	Baixa	Baixa	Média	0 a 1	Baixa	Baixa	Média	0 a 1	Baixa	Baixa	Média
2 a 5	Baixa	Média	Alta	2 a 3	Baixa	Média	Alta	2	Baixa	Média	Alta
6 ou mais	Média	Alta	Alta	4 ou mais	Média	Alta	Alta	3 ou mais	Média	Alta	Alta

FONTE: Adaptado de: IFPUG (2015)

Os pesos atribuídos para realização desta contagem representados na figura acima informam que primeiro se deverá avaliar os tipos de dados para atribuição de pesos: Arquivo Lógico Interno (ALI), Arquivo de Interface Externa (AIE), Saída Externa (SE), Consulta Externa (CE) e Entrada Externa (EE) e o peso corresponde à quantidade de Pontos de Função não ajustados que a função agrega à aplicação, somando-se os pesos de todas as funções, obtém-se o total de PF não ajustados.

FIGURA 45 – PONTOS DE FUNÇÃO NÃO AJUSTADOS POR TIPO DE COMPLEXIDADE DE FUNÇÃO

Tipo de função	Complexidade funcional		
	Baixa	Média	Alta
Entrada	3	4	6
Saída	4	5	7
Consulta	3	4	6
Arquivo interno	7	10	15
Arquivo externo	5	7	10

FONTE: IFPUG (2015)

Para tornar a contagem como ponto de função ajustado deve-se calcular o fator de ajuste avaliando-se quatorze níveis de influências. A cada uma atribuir as características da figura a seguir, é levantado um peso de 0 a 5 do menor ao maior grau de influência.

FIGURA 46 – NÍVEL E INFLUÊNCIA

Características Gerais do Sistema	
[01] Comunicação de Dados [02] Processamento Distribuído [03] Performance [04] Configuração Altamente Utilizada [05] Taxa de Transações [06] Entrada de Dados on-line [07] Eficiência do Usuário Final [08] Atualização on-line [09] Complexidade de Processamento [10] Reutilização [11] Facilidade de Instalação [12] Facilidade de Operação [13] Múltiplas Localidades [14] Facilidade de Mudanças	O próprio guia do FPA sugere níveis de influência: 0 - sem influência. 1 - mínima. 2 - moderada. 3 - média. 4 - significativa. 5 - grande

FONTE: Adaptado de: IFPUG (2015)



Para maiores informações referentes aos pesos dados a cada um dos quatorze níveis de influências, acesse o link <http://www.icmc.usp.br/CMS/Arquivos/arquivos_ enviados/BIBLIOTECA_113_RT_105.pdf>, onde estão descritos entre as páginas 28 a 35.

Para cada característica geral do sistema é feito o cálculo do nível de influência total somando-se os pesos atribuídos à cada característica aplicando a fórmula de cálculo do fator de ajuste: $VAF = (TDI * 0,01) + 0,65$ corresponde a variação de +- 35%.

Após se obter o Ponto de Função Bruto (PFB) deverá calcular os Pontos de Função Ajustados (PFA) utilizando a seguinte fórmula: $PFA = PFB * VAF$. Uma vez que o FPA do projeto tenha sido calculado, o esforço total será calculado multiplicando-se a APF pelo índice de produtividade (IP) da equipe. Esse índice deve ser calculado para ambiente local, e pode variar muito em função do ambiente de trabalho, experiência da equipe e outros fatores. Assim, o esforço (E) e o custo (C) total do projeto poderão ser calculados: $E = APF * IP$ e $C = E * Custo$ (horas).



O site <www.fatto.cs.com.br/editais.asp> armazena editais brasileiros de contratação de software onde a medida de custo é o ponto de função. No site, o preço por ponto de função varia de 100 a 1000 reais, a maioria ficando entre 400 e 600 reais, o que pode ser explicado pelo tipo de sistema que se está contratando.

4.4 PONTOS DE CASO DE USO (PCU)

A técnica de Pontos de Caso de Uso surgiu em 1993 a partir da tese de Gustav Karner. O método é baseado em Análise de Pontos de Função, especialmente MK II, que é um modelo relativamente mais simples que o do IFPUG (WAZLAWICK, 2013).

O método se baseia na análise da qualidade e complexidade dos atores e casos de uso, o que gera pontos de caso de uso não ajustados. Depois, a aplicação e fatores técnicos e ambientais leva aos pontos de uso ajustados.

O processo de contagem dessa métrica consiste nos seguintes passos (HEIMBERG, 2005 apud MEDEIROS, 2004):

1. Relacionar os atores, classificá-los de acordo com seu nível de complexidade (simples, médio ou complexo) atribuindo respectivamente os pesos 1, 2 ou 3, conforme a tabela a seguir. Calcule o TPNAA (Total de Pesos não Ajustados dos Atores) somando os produtos da quantidade de atores pelo seu peso.

TABELA 3 – COMPLEXIDADE DO ATOR

Complexidade do ator	Descrição	Peso
Simples	Poucas entidades de Banco de Dados envolvidas e sem regras de negócio complexas	1
Médio	Poucas entidades de Banco de Dados envolvidas e com algumas regras de negócio complexas	2
Complexo	Regras de negócios complexas e muitas entidades de Bancos de Dados presentes	3

FONTE: Medeiros (2004)

2. Contar os casos de uso e atribuir o grau de complexidade sendo a complexidade baseada no número de classes e transações. Calcule o TPNAUC (Total de Pesos não ajustados dos casos de usos) somando os produtos da quantidade de casos de usos pelo respectivo peso conforme a tabela a seguir.

TABELA 4 – CLASSIFICAÇÃO DOS CASOS DE USO

Tipo de Caso de Uso	Descrição	Peso
Simples	Considerar até 3 transações com menos de 5 classes de análise	5
Médio	Considerar de 4 a 7 transações com 5 a 10 classes de análise	10
Complexo	Considerar de 7 transações com pelo menos 10 classes de análise	15

FONTE: Medeiros (2004)

3. Calcular PCUs não ajustados, também chamados de PCUNA, de acordo com a seguinte fórmula: $PCUNA = TPNAA + TPNAUC$
4. Determinar o fator de complexidade técnica. Os fatores de complexidade técnica variam numa escala de 0 a 5, de acordo com o grau de dificuldade do sistema a ser construído. O valor 0 indica que o grau não está presente ou não é influente, o valor 3 indica influência média e o valor 5 indica influência significativa através de todo o processo. Após determinar o valor dos fatores, multiplicar pelo respectivo peso ilustrado na tabela a seguir, somar o total e aplicar a seguinte fórmula: Fator de complexidade técnica (FCT) = $0.6 + (0.01 * \text{Somatório do Fator técnico})$.

TABELA 5 – FATORES DE COMPLEXIDADE TÉCNICA

Descrição	Peso
Sistemas Distribuídos	2,0
Desempenho da aplicação	1,0
Eficiência do usuário final (on-line)	1,0
Processamento interno complexo	1,0
Reusabilidade do código em outras aplicações	1,0
Facilidade de instalação	0,5
Usabilidade (facilidade operacional)	0,5
Portabilidade	2,0
Facilidade de manutenção	1,0
Concorrência	1,0
Características especiais de segurança	1,0
Acesso direto para terceiros	1,0
Facilidades especiais de treinamento	1,0

FONTE: Medeiros (2004)

5. Determinar o fator de complexidade ambiental: os fatores de complexidade ambientais indicam a eficiência do projeto e estão relacionados ao nível de experiência dos profissionais. Esses fatores descritos na tabela a seguir são determinados através da escala de 0 a 5, onde 0 indica baixa experiência, 3 indica média experiência e 5 indica alta experiência. Após determinar o valor de cada fator, multiplicar pelo peso e somar o total dos valores. Em seguida, aplicar a seguinte fórmula:
6. Fator de complexidade ambiental (FCA) = $1,4 + (-0,03 * \text{Somatório do Fator Ambiental})$
7. Calcular os PCUs ajustados: esse cálculo é realizado com base na multiplicação dos PCU não ajustados, na complexidade técnica e na complexidade ambiental através da seguinte fórmula: $\text{PCUA} = \text{PCUNA} * \text{Fator de complexidade técnica} * \text{Fator de complexidade ambiental}$.

TABELA 6 – FATORES DE COMPLEXIDADE AMBIENTAL

Fator	Descrição	Peso
F1	Familiaridade com o processo de desenvolvimento de software	1,5
F2	Experiência na aplicação	0,5
F3	Experiência com OO, na linguagem e na técnica de desenvolvimento	1,0
F4	Capacidade do líder de análise	0,5
F5	Motivação	1,0
F6	Requisitos estáveis	2,0
F7	Trabalhadores com dedicação parcial	-1,0
F8	Dificuldade da linguagem de programação	-1,0

FONTE: Medeiros (2004)

8. Calcular a estimativa de horas de programação. Karner, o criador da estimativa, sugere a utilização de 20 pessoas-hora por unidade de PCU. Schneider e Winters (apud Heimberg, 2005), sugerem o seguinte refinamento:

X = total de itens de F1 a F6 com pontuação abaixo de 3

Y = total de itens de F7 a F8 com pontuação acima de 3

Se $X + Y \leq 2$, usar 20 como unidade de homens/hora. Se $X + Y = 3$ ou $X + Y = 4$, usar 28 como unidade de homens/hora. Se $X + Y \geq 5$, deve-se tentar modificar o projeto de forma a baixar o número, pois o risco de insucesso é relativamente alto.

Estimativa de horas = PCUA * pessoas hora por unidade de PCU

4.5 MODELO COCOMO II

O modelo construtivo de custo com o nome COCOMO (COnstructive COst MOdel) original tornou-se um dos modelos de estimativa de custo de *software* mais amplamente usado e discutido na indústria. Ele evoluiu para o modelo de estimativa mais abrangente, chamado COCOMO II. Assim como seu predecessor, o COCOMO II é na realidade uma hierarquia de modelos de estimativas que trata das seguintes áreas:

- Modelo de composição de aplicação: usado durante os primeiros estágios de engenharia de *software*, em que o protótipo das interfaces de usuário, a consideração da interação de *software* e sistema, a avaliação do desempenho e a avaliação da maturidade da tecnologia são de suma importância.
- Modelo de estágio de início do projeto: usado quando os requisitos tiverem sido estabilizados e a arquitetura básica de *software* tiver sido estabelecida.
- Modelo de estágio pós-arquitetura. Usado durante a construção do *software*.

Assim como todos os modelos de estimativas para *software*, os modelos COCOMO II requerem informações de tamanho. Há disponíveis três diferentes opções como parte da hierarquia de modelo: pontos de objeto, pontos de função e linhas de código-fonte (PRESSMAN, 2011). O modelo de composição de aplicação COCOMO II utiliza pontos de objeto e é ilustrado nos próximos parágrafos. Devemos observar que há também disponíveis outros modelos de estimativa mais sofisticados (usando APP ou LOC) como parte do COCOMO II.

TABELA 7 – PESO DA COMPLEXIDADE PARA TIPO DE OBJETO

Tipo de Objeto	Peso da Complexidade		
	Simples	Média	Difícil
Tela	1	2	3
Relatório	2	5	8
Componente 3GL			10

FONTE: Pressman (2011)

Assim como APF, o ponto de objeto é uma medida indireta de *software* calculada por meio de contagens dos números de (1) telas (na interface do usuário), (2) relatórios e (3) componentes que podem ser necessários para construir a aplicação. Cada instância de objeto (por exemplo, uma tela ou um relatório) é classificada em um dentre três níveis de complexidade (simples, médio ou difícil). Essencialmente, a complexidade é uma função da qualidade e origem das tabelas de dados-cliente e servidor necessárias para gerar a tela ou relatório e o número de visualizações ou seções apresentadas como parte da tela ou relatório.

Uma vez determinada a complexidade, os números de telas, relatórios e componentes são ponderados de acordo com a tabela acima. A contagem de pontos de objeto é então determinada multiplicando-se o número original de instâncias de objeto pelo fator de peso na figura e somando para obter o total da contagem de pontos de objeto. Quando deve ser aplicado desenvolvimento baseado em componentes ou reutilização de *software* em geral, é estimada a porcentagem de reutilização (% reúso) e é ajustada a contagem de pontos de objeto: NOP = (pontos de objeto) * [(100 - %reúso)/100]. Em que NOP é definido como novos pontos de objeto.

Para derivar uma estimativa de esforço com base no valor calculado para NOP, deve ser derivada uma “taxa de produtividade”. A tabela abaixo apresenta a taxa de produtividade: PROD = (NOP)/pessoa-mês. Para diferentes níveis de experiência do desenvolvedor e maturidade do ambiente de desenvolvimento.

Uma vez determinada a taxa de produtividade, calcula-se a estimativa de esforço do projeto: Esforço estimado = NOP/PROD.

Em modelos COCOMO II mais avançados, é necessária uma variedade de fatores de escala, custos e procedimentos de ajuste. Uma discussão completa desses tópicos está além do escopo de caderno. Acadêmico(a), se tiver interesse, você pode visitar o *site* do COCOMO II.

TABELA 8 – FATORES DE AJUSTES

Experiência/capacidade do desenvolvedor	Muito baixa	Baixa	Nominal	Alta	Muito alta
Maturidade /capacidade do ambiente	Muito baixa	Baixa	Nominal	Alta	Muito alta
PROD	4	7	13	25	50

FONTE: Pressman (2011)

4.6 ESTIMATIVA PARA PROJETOS ORIENTADOS A OBJETO

É conveniente suplementar os métodos convencionais de estimativa de custo de *software* com uma técnica criada explicitamente para *software* orientado a objeto. Pressman sugere via Lorenz e Kidd as seguintes abordagens:

1. Desenvolva estimativas usando decomposição de esforço, análise de pontos de função (APF) e qualquer outro método para aplicações convencionais.
2. Usando a modelagem UML através da análise de casos de uso, desenvolva casos de uso e determine uma contagem. Reconheça que o número de caso de uso pode mudar à medida que o projeto avança.
3. Como base no modelo de requisito utilizando caso de uso, determine o número de classe-chave (chamadas de classe).
4. Classifique o tipo de interface para a aplicação e desenvolva um multiplicador para classes de apoio:

TABELA 9 – TIPO DE INTERFACE

Tipo de Interface	Multiplicador
Sem GUI	2,0
Interface de Usuário baseada em texto	2,25
GUI	2,5
GUI complexa	3,0

FONTE: Pressman (2011).

Multiplique o número de classes-chave (passo 3) pelo multiplicador para obter uma estimativa do número de classes de apoio.

5. Multiplique o número total de classes (classes-chave + classe de apoio) pelo número médio de unidades de trabalho por classe. Pressman sugere 15 a 20 pessoas-dia por classe.
6. Faça uma verificação cruzada da estimativa baseada em classe multiplicando o número médio de unidades de trabalho por caso de uso.

FONTE: PRESSMAN, R. S. *Engenharia de Software – Uma abordagem profissional*. 7. ed. Porto Alegre: AMGH Editora Ltda., 2011.

LEITURA COMPLEMENTAR

A fim de elucidar melhor este estudo de métricas e estimativas em projetos de *software*, a seguir serão apresentados **três exemplos de cálculos de métricas de software** utilizados por empresas desenvolvedoras de *software*. A primeira baseada em Linha de Código (LOC), a segunda baseada em Análise de Ponto de Caso de Uso e a última baseada em Análise de pontos de Função. O exemplo de LOC foi extraído de Roger Pressman (2011, 612-614 p.). O exemplo da Análise de Ponto de Caso de Uso extraído de Viviane Heimberg (2005) e o último exemplo de Análise de Pontos de Função foi elaborado pelo próprio autor.

1. UM EXEMPLO DE ESTIMATIVA BASEADA EM LINHA DE CÓDIGO (LOC)

Como exemplo desta técnica vamos considerar um pacote de *software* a ser desenvolvido para uma aplicação de projeto auxiliado por computador para componentes mecânicos. O *software* deve ser executado em uma estação de trabalho de engenharia e deve ter interface com vários periféricos gráficos incluindo um *mouse*, um teclado, um monitor colorido de alta resolução e uma impressora. Pode ser formulada uma definição preliminar do escopo do *software*.

O *software* CAD mecânico aceitará dados geométricos bidimensionais e tridimensionais fornecidos por um engenheiro. O engenheiro vai interagir e controlar o sistema CAD através de uma interface de usuário que irá exibir características de um bom *design* de interface homem/máquina. Todos os dados geométricos ou outras informações de suporte serão mantidos em uma base de dados CAD. Serão desenvolvidos módulos de análise de projeto para produzir a saída requerida, a ser exibida em uma variedade de dispositivos gráficos. O *software* será projetado para controlar e interagir com dispositivos periféricos que incluem um *mouse*, um teclado, impressora *laser* e um ploter.

Essa definição de escopo é preliminar – não é delimitada. Cada sentença deverá ser expandida para proporcionar detalhe concreto e limites quantitativos. Por exemplo, antes de iniciar a estimativa, o planejamento deve determinar o que significa “características de projeto de uma boa interface homem/máquina” ou qual deverão ser o tamanho e a sofisticação da “base de dados CAD”.

Para nosso propósito, suponha que ocorreu um refinamento adicional e que as principais funções do *software* listadas na tabela abaixo estão identificadas. Seguindo a técnica de decomposição por linha de código, é desenvolvida uma tabela de estimativas. Para cada função é desenvolvido um intervalo de estimativa LOC. Por exemplo, o intervalo de estimativas LOC para a função de análise geométrica 3-D é otimista para 4.600 LOC; mais esperada para 6.900 LOC; e pessimista para 8.600 LOC. Aplicando a equação $KSLOC = (4*KSLOC \text{ esperado} + KSLOC \text{ otimista} + KSLOC \text{ pessimista})/6$, ou seja, $KSLOC = (4*6.900 \text{ LOC} + 4.600 \text{ LOC} + 8.900 \text{ LOC})/6$. O valor esperado para a função de análise geométrica 3D é 6.800. Outras estimativas são obtidas de forma semelhante.

Somando verticalmente na coluna de estimativa LOC, obtém-se uma estimativa de 33.200 linhas de código para o sistema CAD.

Um exame de dados históricos indica que a produtividade média organizacional para sistemas deste tipo é de 620 LOC/pm. Com base em um valor bruto de mão de obra de R\$ 8 mil por mês, o custo por linha de código é de aproximadamente R\$ 13. Com base na estimativa LOC e dados históricos de produtividade, o custo total estimado do projeto é de R\$ 431 mil e o esforço estimado é de 54 pessoas-mês.

TABELA 10 – CONTAGEM FCU

FUNÇÃO	LOC estimado
Interface de usuário e recurso de controle	2.300
Análise geométrica bidirecional	5.300
Análise geométrica tridimensional	6.800
Gerenciamento de base de dados	3.350
Recursos de visualização da computação gráfica	4.950
Função de controle de periféricos	2.100
Módulo de análise do projeto	8.400
Linha de Código estimada	33.200

FONTE: Pressman (2011).

2. UM EXEMPLO DE ESTIMATIVA BASEADA EM ANÁLISE DE PONTOS DE CASO DE USO

Foram analisados os diagramas de casos de uso dos seguintes três projetos: Sistema de cálculo de Folha de Pagamento (Projeto 1), Sistema Contábil (Projeto 2) e o Sistema de Cartão Ponto (Projeto 3). Em cada projeto foram realizadas as fases de concepção e a primeira iteração da fase de elaboração. Na fase de concepção somente foram elaborados os diagramas de nível 0 de apenas 1 módulo em cada projeto. Para a primeira iteração da fase de elaboração foram ampliados apenas alguns casos de uso considerados relevantes pelos analistas de cada projeto.

O índice do fator de complexidade ambiental obteve o mesmo valor para os três projetos pois os analistas possuíam o mesmo grau de experiência em UML, mesma familiaridade com processo unificado de desenvolvimento, mesma experiência em orientação a objetos, mesmo grau de motivação e mesmo conhecimento do ambiente de desenvolvimento, pois todos receberam um treinamento padronizado antes do início dos projetos. Os requisitos foram considerados estáveis pois trata-se da conversão de três sistemas já existentes em ambiente cliente/servidor para o ambiente Web.

O índice do fator técnico do projeto variou apenas nos itens referentes a complexidade de processamento, concorrência e acesso direto a terceiros. Todos os sistemas executam cálculos complexos e precisos, exigem máxima segurança e possuem mais de cem usuários acessando simultaneamente. Os pesos utilizados para os fatores técnicos e ambientais foram os sugeridos por Medeiros (2004).

As equipes dos respectivos projetos ainda não estão completas e a metodologia de desenvolvimento adaptada do processo unificado ainda está em teste, por isso o total de pessoas horas por unidade de PCU considerado foi de 20 horas homem para uma primeira análise. As horas estimadas não foram divididas pelo total de membros da equipe. Os resultados desta pesquisa na fase de concepção e os da fase de elaboração foram ilustrados nas tabelas a seguir.

TABELA 11 – ESTIMATIVAS DA FASE DE CONCEPÇÃO

Projetos	Atores	Casos de Uso	FCT	FCA	PCUNA	PCUA	Horas Estimadas
Projeto 1	4	1	1,00	0,81	22	17,93	358,60
Projeto 2	6	2	1,02	0,81	37	30,76	615,16
Projeto 3	7	2	1,03	0,81	39	32,74	654,77

FONTE: Heimberg (2005)

TABELA 12 – ESTIMATIVAS DE FASE DE ELABORAÇÃO

Projetos	Atores	Casos de Uso	FCT	FCA	PCUNA	PCUA	Horas Estimadas
Projeto 1	4	5	1,00	0,81	72	58,68	1.173,60
Projeto 2	6	8	1,02	0,81	87	72,32	1.446,46
Projeto 3	7	4	1,03	0,81	64	53,72	1.074,50

FONTE: Heimberg (2005)

Verificou-se junto às equipes de desenvolvimento e aos coordenadores dos projetos que os tempos obtidos pela estimativa estavam muito acima dos obtidos em projetos semelhantes. Os coordenadores dos projetos identificaram que 20 horas/homem por total de tempo de unidade de PCU era um número muito alto e não representava corretamente uma boa média para todos os tipos de tempos por nível de complexidade de casos de uso. A metodologia de desenvolvimento da empresa utiliza uma camada de código que abstrai grande parte da geração de código básica dos seus sistemas, reduzindo a quantidade de horas/homem para realizar o desenvolvimento de casos de uso para aproximadamente 5 horas/homem em casos de uso simples, 9 horas/homem para casos de uso médios e 24 horas/homem para casos de uso complexos.

Foi decidido realizar uma nova estimativa nos três projetos utilizando os casos de uso da fase de elaboração por estar mais completa, desta vez ajustando-se os pesos dos casos de uso para simples=5, médio=10, complexo = 25 e modificar a quantidade de horas/homem para uma média de 10 horas/homem.

Os resultados obtidos foram os seguintes:

TABELA 13 – ESTIMATIVAS DA FASE DE ELABORAÇÃO COM OS NOVOS PESOS AJUSTADOS

Projetos	Atores	Casos de Uso	FCT	FCA	PCUNA	PCUA	Horas Estimadas
Projeto 1	4	5	1,00	0,81	54	43,74	437,40
Projeto 2	6	8	1,02	0,81	76	61,50	615,60
Projeto 3	7	4	1,03	0,81	57	46,17	461,70

FONTE: Heimberg (2005)

3 UM EXEMPLO DE ESTIMATIVA BASEADA EM ANÁLISE DE PONTOS DE FUNÇÃO (APF)

Calcule o custo do sistema seguinte, levando em consideração que em cada hora de trabalho consegue-se realizar dois pontos de função e que cada hora/pessoa custa R\$ 8,00 (oito reais). Um levantamento de dados realizado por analistas de sistemas apresentou os seguintes resultados em relação a um **Sistema de Gerenciamento de Leitos Hospitalares**: as informações internas do sistema estão agrupadas no cadastro de diagnósticos (18 itens de dados), de leitos (5 itens de dados) e de remédios (3 registros lógicos e 23 itens de dados). As informações externas a serem utilizadas pelo sistema são: cadastro de médicos (31 itens de dados) e de pacientes (16 itens de dados). Os principais relatórios do sistema são de histórico do paciente (4 arquivos referenciados e 10 itens de dados) e de ocupação de leitos (2 arquivos referenciados e 5 itens de dados). As principais entradas do sistema são: incluir, excluir e alterar os dados dos cadastros internos. O sistema permite consulta da disponibilidade de remédios, disponibilidade de leitos e de horários de trabalho dos médicos. Todas as consultas são de complexidade média. A comunicação de dados é crítica, portanto de grande influência. O volume de transações e a eficiência do usuário final possuem significativa influência. Há uma preocupação moderada em reutilização de código. O sistema será utilizado em vários ambientes de *hardware* e *software*, portanto múltiplos locais possuem influência significativa. O restante das características terá influência mínima. Qual é o ALI, AIE, CE, EE, SE, PFB, FA (NI), PFA e o Custo do Sistema.

Resolução

Cada hora de trabalho = 2 pontos de função
 Hora/Pessoa = M\$ 8,00

FIGURA 47 – CONTAGEM DOS ELEMENTOS

ALI e AIE		SE e CE			EE							
Tipo de Registro (TR)	Tipo de Dados (TD)			Tipo de Dados (TD)			Tipo de Dados (TD)					
	<20			<6			<5					
	1	Baixa	Baixa	Média	<2*	Baixa	Baixa	Média	<2	Baixa	Baixa	Média
	2-5	Baixa	Média	Alta	2-3	Baixa	Média	Alta	2	Baixa	Média	Alta
>5	Média	Alta	Alta	>3	Média	Alta	Alta	>2	Média	Alta	Alta	
OU												
Classes RET		Argumentos DET			Argumentos DET			Argumentos DET				
Classes RET	1 a 19	20 a 50	51 ou mais	Classes FTR	1 a 5	6 a 19	20 ou mais	Classes FTR	1 a 4	5 a 15	16 ou mais	
1	Baixa	Baixa	Média	0 a 1	Baixa	Baixa	Média	0 a 1	Baixa	Baixa	Média	
2 a 5	Baixa	Média	Alta	2 a 3	Baixa	Média	Alta	2	Baixa	Média	Alta	
6 ou mais	Média	Alta	Alta	4 ou mais	Média	Alta	Alta	3 ou mais	Média	Alta	Alta	

FONTE: Adaptado de: IFPUG (2015)

FIGURA 48 – PONTOS DE FUNÇÃO NÃO AJUSTADOS POR TIPO DE COMPLEXIDADE DE FUNÇÃO

Tipo de função	Complexidade funcional		
	Baixa	Média	Alta
Entrada	3	4	6
Saída	4	5	7
Consulta	3	4	6
Arquivo interno	7	10	15
Arquivo externo	5	7	10

FONTE: IFPUG (2015)

1. Pontos de Função Não Ajustados

Para calcular o valor de pontos de função não ajustados, primeiro se deve contar para cada tipo de elemento (Figura 47 = ALI, AIE, SE, CE e EE) a quantidade de registro (ALI e AIE) ou quantidade de arquivos (SE, CE e EE).

Veja neste primeiro exemplo do diagnóstico a seguir, ele tem 1 registro e 18 campos ao qual corresponde ao tipo **BAIXA**. Depois que utilizar a Figura 47 e saber que é de complexidade Baixa, basta consultar a Figura 48 e saber qual é o valor para complexidade baixa para o tipo de Arquivo interno chegando ao valor 7 e assim por diante para cada um destes valores abaixo. Obs.: estas tabelas das Figuras 47 e 48 são padrão de uso na APF.

Arquivos Lógicos Internos

Diagnósticos (1,18) Simples 7

Leitos (1,5) Simples 7

Remédios (3,23) Média 10

Arquivos Lógicos Internos => $2 \times 7 + 1 \times 10 = 24$

Arquivos de Interface Externa

Médicos (1, 31) Simples 5

Pacientes (1,16) Simples 5

Arquivos de Interface Externa => $2 \times 5 = 10$

Entradas Externas

Incluir Diagnósticos (1,18) Simples 3

Incluir Leitos (1,5) Simples 3

Incluir Remédios (3,23) Simples 3

Excluir Diagnósticos (1,18) Simples 3

Excluir Leitos (1,5) Simples 3

Excluir Remédios (3,23) Simples 3

Alterar Diagnósticos (1,18) Simples 3

Alterar Leitos (1,5) Simples 3

Alterar Remédios (3,23) Simples 3

Entradas Externas => $9 \times 3 = 27$

Saídas Externas

Histórico do Paciente (4,10) Complexa 7

Ocupação de Leitos (2,5) Simples 4

Saídas Externas => $1 \times 7 + 1 \times 4 = 11$

Consultas Externas

Disponibilidade de Remédios Média 4

Disponibilidade de Leitos Média 4

Horário de Trabalho dos Médicos Média 4

Consultas Externas => $3 \times 4 = 12$

Total de Pontos de Função Não Ajustados (somas do total de cada tipo coletado acima)

Arquivos Lógicos Internos 24

Arquivos de Interface Externa 10

Entradas Externas 27

Saídas Externas 11

Consultas Externas 12

TOTAL GERAL: 84

2. Fator de Ajuste

FIGURA 49 – NÍVEL E INFLUÊNCIA

Características Gerais do Sistema	
[01] Comunicação de Dados [02] Processamento Distribuído [03] Performance [04] Configuração Altamente Utilizada [05] Taxa de Transações [06] Entrada de Dados on-line [07] Eficiência do Usuário Final [08] Atualização on-line [09] Complexidade de Processamento [10] Reutilização [11] Facilidade de Instalação [12] Facilidade de Operação [13] Múltiplas Localidades [14] Facilidade de Mudanças	O próprio guia do FPA sugere níveis de influência: 0 - sem influência. 1 - mínima. 2 - moderada. 3 - média. 4 - significativa. 5 - grande

FONTE: Adaptado de: IFPUG (2015)

Características Nível de influência Peso

1. Comunicação de Dados: Grande influência - 5
2. Processamento Distribuído: Nenhuma influência - 1
3. Performance: Nenhuma influência - 1
4. Utilização de Equipamento: Nenhuma influência - 1
5. Volume de Transações: Influência significativa - 4
6. Entrada de Dados "on-line": Nenhuma influência - 1
7. Eficiência do Usuário Final: Influência significativa - 4
8. Atualização "on-line": Nenhuma influência - 1
9. Processamento Complexo: Nenhuma influência - 1
10. Reutilização de Código: Influência mínima - 2
11. Facilidade de Implantação: Nenhuma influência - 1
12. Facilidade Operacional: Nenhuma influência - 1
13. Múltiplos Locais: Influência significativa - 4
14. Facilidade de Mudanças: Nenhuma influência - 1

TOTAL 28

$$\text{Fator de Ajuste} = (NI * 0,01) + 0,65$$

$$\text{Fator de Ajuste} = (28 * 0,01) + 0,65$$

$$\text{Fator de Ajuste} = 0,93$$

3. Pontos de Função Ajustados

$$\text{PF Ajustados} = \text{PF não ajustados} * \text{Fator de Ajuste}$$

$$\text{PF Ajustados} = 84 * 0,93$$

$$\text{PF Ajustados} = 78,12$$

Custo do Sistema

$$\text{Custo do sistema} = (\text{PF Ajustados} / \text{Pontos de função por hora}) * \text{Custo hora/pessoa}$$

$$\text{Custo do sistema} = (78,12/2) * 8$$

$$\text{Custo do sistema} = R\$ 312,48$$

RESUMO DO TÓPICO 2

Neste tópico, você aprendeu que:

- Estimar *software* significa determinar quanto de dinheiro, esforço, recursos e tempo serão necessários para criar um sistema, primeiro é feita a avaliação do escopo para saber a complexidade do trabalho a ser realizado e a partir do perfil dos profissionais que irão trabalhar no projeto irá determinar o tempo de execução do projeto.
- O PMBOK descreve que para garantir a aplicabilidade da gestão de custos em projetos os processos de Planejar Gerenciamento dos Custos, Estimar Custos, Definir Orçamento e Controlar Custos são aplicados utilizando-se processos de entradas, ferramentas e técnicas e as saídas.
- As entradas são as informações ou dados obtidos referentes ao projeto. As ferramentas e técnicas podem ser um padrão que formaliza os procedimentos a serem adotados para efetiva administração dos custos do projeto e as saídas são produtos, como as estimativas de custos, recursos, cronogramas, tempo das atividades, previsões orçamentárias, entre outros.
- Através de Métricas de *Software* pode-se identificar a quantidade de esforço, de custo e das atividades que serão necessárias para a realização do projeto. É a medição de um atributo (propriedades ou características) de uma determinada entidade (produto, processo ou recursos). Exemplos: Tamanho do produto em número de linhas de código, número de pessoas, esforço, tempo, custo para a realização de uma tarefa etc.
- Os principais motivos ao utilizar a métrica de *software* é oferecer a possibilidade de entender e aperfeiçoar o processo de desenvolvimento, melhorar a gerência de projetos e o relacionamento com clientes, reduzir frustrações e pressões de cronograma, indicar a qualidade de um produto de *software*, avaliar a produtividade do processo, melhorar a exatidão das estimativas etc.
- Existem vários métodos que podem ser utilizados para se estimar o custo do desenvolvimento e a vida útil de um sistema, entre eles: Linhas de Código (LOC); Pontos de História; Análise de Pontos de Função e Análise de Pontos de Caso de Uso, COCOMO II e estimativa para projetos Orientados a Objeto.
- Linha de código (LOC): é métrica orientada ao tamanho do *software*, onde e consiste em estimar o número de linhas que um programa deverá ter. Para sua definição são considerados três valores, o LOC otimista, o LOC pessimista e o LOC esperado e a partir dos dados coletados seus valores são aplicados na fórmula: $KSLOC = (4 * KSLOC \text{ esperado} + KSLOC \text{ otimista} + KSLOC \text{ pessimista}) / 6$.

- Pontos de História: é uma métrica de estimativa de tempo, pergunta-se à equipe quanto tempo tantas pessoas que se dedicassem a uma história de usuário levaria para terminá-la, gerando uma versão executável funcional. Então multiplica o número de pessoas pelo número de dias para chegar ao valor de pontos de história.
- Análise de Pontos de Função (APF) define processos e técnicas formais e padronizadas para dimensionamento e estimativa da complexidade de sistema, ou seja, para medir o tamanho do escopo.
- Para manter e determinar os procedimentos de contagem APF segue-se os seguintes passos: (1) Determinar o tipo de contagem (desenvolvimento, melhoria ou aplicação existente). (2) Determinar as Fronteiras da aplicação (escopo do sistema). (3) Identificar e atribuir valor em pontos de função não ajustados para as transações sobre dados (entrada, consultas e saídas externas). (4) Identificar e atribuir valor em pontos de função não ajustados (UFPA) para os dados estáticos (arquivos internos e externos). (5) Determinar o valor de ajuste técnico (VAF) e (6) Calcular o número de pontos de função ajustados (AFP).
- Num primeiro momento define o tipo de contagem que pode ser estimado ou detalhado. Após isso são definidos os tipos de dados, e o FPA avalia duas naturezas dos dados. Os dados estáticos são a representação estruturada dos dados, na forma de arquivos internos ou externos e os dados dinâmicos são a representação das transações sobre os dados na forma de entradas, saídas e consultas externas.
- Para obter o FPA não ajustado para cada função de dados são contados os tipos de registros (RET) e a quantidade de campos de dados (DET) que ele possui e para cada função de transação são contados os arquivos referenciados (FTR) e a quantidade de campos de dados destes arquivos.
- Para obter o PFA ajustado deve-se calcular o fator de ajuste com quatorze níveis de ajustes onde a cada um é atribuído peso de 0 a 5 do menor para o maior grau de influência. Uma vez que o FPA do projeto tenha sido calculado, o esforço total será calculado multiplicando-se a APF pelo índice de produtividade (IP) da equipe chegando-se aos valores do esforço e do custo do projeto. $E = APF * IP$ e $C = E * \text{custo (hora)}$.
- Pontos de Caso de Uso (PUC): baseia na análise da qualidade e complexidade dos atores e casos de uso, o que gera pontos de caso de uso não ajustados. Depois, a aplicação e fatores técnicos e ambientais levam aos pontos de caso de uso ajustados.
- Primeiro deve relacionar os atores, classificá-los de acordo com seu nível de complexidade (simples, médio ou complexo) atribuindo respectivamente os pesos 1, 2 ou 3. Em seguida contar os casos de uso e atribuir o grau de complexidade sendo baseada no número de classe e transações.

- Utiliza-se a seguinte fórmula para calcular PCUs não ajustados: $PCUNA = TPNAA + TPNAUC$ e para chegar ao PUCs ajustado determina-se o fator de complexidade técnica que varia da escala de 0 a 5 para cada grau de dificuldade do sistema a ser construído e por fim chega-se ao valor PUCs ajustado utilizando a seguinte fórmula: $PCUA = PCUNA * \text{Fator de complexidade técnica} * \text{Fator de complexidade ambiental}$. E para 8. Calcular a estimativa de horas de programação basta multiplicar o PCUs ajustado pelo número pessoa hora por unidade de PCU.
- Modelo COCOMO II: é um modelo construtivo de custo que trata das seguintes áreas: Modelo de composição de aplicação, Modelo de estágio de início do projeto e Modelo de estágio pós-arquitetura.
- Para requerer informações de tamanho como parte da hierarquia do projeto há disponíveis três diferentes opções: pontos de objeto, pontos de função e linhas de código-fonte.
- A contagem de pontos de objeto é então determinada multiplicando-se o número original de instâncias de objeto pelo fator de peso e somando para obter o total da contagem de pontos de objeto. Quando deve ser aplicado desenvolvimento baseado em componentes ou reutilização de *software* em geral, é estimada a porcentagem de reutilização (% reúso) e é ajustada a contagem de pontos de objeto: $NOP = (\text{pontos de objeto}) * [(100 - \% \text{ reúso})/100]$. Em que NOP é definido como novos pontos de objeto.
- Para derivar uma estimativa de esforço com base no valor calculado para NOP, deve ser derivada uma “taxa de produtividade”. A tabela abaixo apresenta a taxa de produtividade: $PROD = (NOP)/\text{pessoa-mês}$. Para diferentes níveis de experiência do desenvolvedor e maturidade do ambiente de desenvolvimento. Uma vez determinada a taxa de produtividade, calcula-se a estimativa de esforço do projeto: Esforço estimado = $NOP/PROD$.
- Estimativa para projetos Orientados a Objeto: utiliza qualquer método anterior para decomposição de esforço, utilizando a modelagem UML através da PCU, determine o número de classe-chave e classifique o tipo de interface para a aplicação e desenvolva um multiplicador para classes e multiplique o número das classes-chave pelo multiplicador para obter uma estimativa do número de classes de apoio. Multiplique o número total das classes (classes-chave + classe de apoio) pelo número médio de unidades de trabalho por classe.

AUTOATIVIDADE



- 1 Acerca do tema estimativas e métricas de projetos de *software* analise as sentenças abaixo e assinale com V para verdadeiro ou F para Falso:
 - a) () Estimar *software* significa determinar quanto de dinheiro, esforço, recursos e tempo serão necessários para criar um sistema.
 - b) () O esforço de um projeto de *software* não está relacionado à produtividade, que é medida pela qualidade de trabalho realizada pela equipe.
 - c) () Uma forma de estimar a produtividade é, por exemplo, a quantidade de linha de código pessoa-mês.
 - d) () Em muitas situações, para adequar o ritmo do desenvolvimento às estimativas, a qualidade é sacrificada deixando as coisas ainda piores.
 - e) () A estimativa de tamanho de um projeto de *software* não tem impacto na solução técnica do projeto pois sua duração é só no início do projeto.
- 2 Quais são os quatro processos utilizados para o gerenciamento de custos de projetos conforme definido pelo guia do PMBOK 5^a edição escrita pelo PMI?
- 3 O gerenciamento de custos de projetos é realizado através de três itens a saber:

I. as Entradas.

II. as Ferramentas e Técnicas.

III. as Saídas.

- () Um padrão que formaliza os procedimentos a serem adotados para efetiva administração dos custos do projeto.
- () São produtos, como as estimativas de custos, recursos, cronogramas, tempo das atividades, previsões orçamentárias, entre outros
- () As informações ou dados obtidos referentes ao projeto, oriundos de fatores internos ou externos (pessoas, instalações, equipamentos, materiais etc.).

De acordo com as sentenças acima, associe a sequência correta das definições:

- a) () I – II – III.
- b) () II – III – I.
- c) () II – I – III.
- d) () II – III – I.

- 4 Defina o que é métrica de *software*.
- 5 Seguem os principais métodos que podem ser utilizados para se estimar o desenvolvimento e a vida útil de um sistema:
- I. Linhas de Código (LOC);
 - II. Pontos de História;
 - III. Análise de Pontos de Função;
 - IV. Análise de Pontos de Caso de Uso;
 - V. COCOMO II;
 - VI. Estimativa para projetos Orientados a Objeto.
- () Convém de métodos de estimativa de custo de *software* com uso de técnica criada explicitamente para *software* orientado a objeto.
- () É possível quantificar as funções de um sistema considerando aspectos significantes para o usuário, portanto, irá considerar os requisitos de negócio que o sistema atende e principalmente que sua contagem é independente de tecnologia
- () Métrica de estimativa de tempo, onde é a estimativa de esforço preferida de métodos ágeis como Scrum e XP, relativa à equipe de desenvolvimento.
- () É um modelo relativamente mais simples, baseia na análise da qualidade e complexidade dos atores e casos de uso.
- () Foi possivelmente a primeira a surgir e consiste em estimar o número de linhas que um programa deverá ter, normalmente a partir da opinião de especialistas e histórico de projetos passados.
- () Como todos os modelos de estimativas para *software*, requerem informações de tamanho em três diferentes opções como parte da hierarquia de modelo: pontos de objeto, pontos de função e linhas de código-fonte.

De acordo com as sentenças acima, associe a sequência correta das definições de cada um dos tipos de métricas de *software*:

- a) () VI – II – III – I – IV – V.
- b) () I – III – V – IV – VI – II.
- c) () VI – III – II – IV – I – V.
- d) () V – II – III – IV – I – VI.

GERÊNCIA DE CONFIGURAÇÃO DE SOFTWARE

1 INTRODUÇÃO

Durante todo o ciclo de vida de projetos e produtos de *software*, ocorrem muitas alterações, portanto, a Gerência de Configuração de *Software* (GCS) é uma atividade do tipo guarda-chuva aplicada essencialmente para manter o desenvolvimento de *software* controlável.

As alterações acontecem por conta de mudanças de requisitos, mudanças no entendimento dos usuários sobre suas necessidades, aspectos legais, variados motivos que fazem com que ocorra aumento de alterações no ciclo de vida de um projeto, e consequentemente aumenta-se exponencialmente o risco de ocorrerem defeitos no produto final.

No desenvolvimento de sistemas, os artefatos de um projeto são mantidos, guardados e atualizados através de um repositório padrão (diretório) através da GCS que define critérios que permitem realizar tais modificações mantendo-se a consistência e a integridade do *software* com as especificações permitindo manter a estabilidade na evolução do projeto (PRESSMAN, 2011).

Segundo Ferreira (2015), a GC permite minimizar os problemas decorrentes do processo de desenvolvimento, através de um controle sistemático sobre as modificações. Não é objetivo evitar modificações, mas permitir que elas ocorram sempre que possível, sem que ajam falhas inerentes ao processo.

A partir deste entendimento Ferreira (2015) nos reforça que a GCS atua como um **suporte** sobre o qual as fases do desenvolvimento são conduzidas e os produtos controlados. Porém, sua prática é aplicada apenas quando existe um processo de desenvolvimento bem definido, com atividades agrupadas em fases, constituídas por objetivos bem definidos e documentados através de normas, ferramentas e templates que permitam gerenciar de maneira satisfatória os itens de configuração de um sistema.

A Gerência de Configuração de *Software* é solução modularizada que provê a estabilidade dos ambientes de *software* e o controle de arquivos. Pesquisas mostram que entre 15% a 20% do tempo de desenvolvimento de *software* são consumidos com atividades de configuração e gestão de mudanças, e este percentual pode aumentar quando se trata de manutenção (PRESSMAN, 2011).

A GCS propõe automação, versionamento, gerência de mudanças e gerência de *releases*. Seus benefícios são: redução dos custos de desenvolvimento, o aumento de colaboração entre equipes, ganho de tempo nas tarefas repetitivas e garantia da integridade dos ativos de desenvolvimento.

2 ITENS DE CONFIGURAÇÃO

A definição de Configuração consiste em que a configuração de um sistema é uma coleção de versões específicas de ITENS DE CONFIGURAÇÃO (*hardware* ou *software*) que são combinados de acordo com procedimentos específicos de construção para servir a uma finalidade particular.

Roger Pressman (2011) explica que “item de configuração” é cada um dos elementos de informação que são criados durante o desenvolvimento de um produto de *software*, ou que para este desenvolvimento sejam necessários, que são identificados de maneira única e cuja evolução é passível de rastreamento. Ou seja, tanto os documentos como os arquivos-fonte que compõem um produto de *software* são itens de configuração (IC), assim como também o são as ferramentas de *software* necessárias para o desenvolvimento.

Qualquer sistema em desenvolvimento deve ser particionado em itens de configuração, e o seu desenvolvimento é visto como o desenvolvimento de cada um dos ICs. Cada IC, por sua vez pode ser particionado em outro conjunto de ICs e assim sucessivamente, até que se resulte em um conjunto de ICs não particionáveis, que são desenvolvidos segundo um ciclo de vida propriamente definido (FERREIRA, 2015).

Na prática o processo de desenvolvimento de *software* é composto das fases de iniciação, elaboração, construção e transição onde em cada fase são criados diversos artefatos no projeto, nas GCs estes artefatos precisam ser mapeados e estarem sob IC. Tal conjunto representa um estágio do desenvolvimento, o qual é passível de modificações apenas mediante um mecanismo formal de alterações. A este conjunto é dado o nome de *Baselines*.

Pressman (1995) apresenta a seguir IC alvo das técnicas de GCS e formam um conjunto de linhas básicas: especificação do *software*, plano de projeto, requisitos, protótipos, manuais, fluxo de dados, DER, MER, código fonte, plano de testes, executável, BD, documentação de manutenção etc.

O desenvolvimento com configurações base pode, então, ser resumido nos seguintes pontos:

- Caracterização do ciclo de vida, identificando-se as fases pelas quais o desenvolvimento do *software* irá passar e, dentro delas, as atividades a serem realizadas e os produtos a serem desenvolvidos.
- Definição do conjunto de *baselines*. Para cada *baseline* planejada, deve-se estabelecer quais serão os ICs que a irão compor e quais as condições impostas para seu estabelecimento.
- *Baselines* representam marcos no processo de desenvolvimento: uma nova *baseline* é estabelecida no final de cada fase do ciclo de vida do *software*.
- Durante cada fase, o desenvolvimento dos ICs a ela referentes está sob total controle de seus desenvolvedores, e realiza-se com ampla liberdade, podendo os ICs serem criados e modificados com bastante facilidade.
- Durante cada fase, entretanto, a modificação de uma configuração base anteriormente estabelecida somente pode ser feita de forma controlada, mediante um processo bem definido.
- Ao ser estabelecida, cada *baseline* incorpora integralmente a anterior. Desta forma, em qualquer instante do desenvolvimento, a última *baseline* estabelecida representa o estado atual do desenvolvimento como um todo.
- O estabelecimento de cada *baseline* somente é realizado após ser aprovada por procedimentos de consistência interna, verificação e validação.

FONTE: FERREIRA, Gladstone. Gerência de Configuração. Disponível em: <<http://www.cin.ufpe.br/~gfn/qualidade/gc.html>>. Acesso em: 23 ago. 2015.

Em outras palavras, a configuração é o estado do conjunto de itens que formam o sistema em um determinado momento; e a GCS é o controle da evolução dessas configurações durante o ciclo de vida do projeto. A GCS é extremamente útil e importante. Por isso, faz parte de modelos importantes de maturidade de processo de desenvolvimento, tais como o CMMI, MPSBr e o SPICE (DIAS, 2015).

3 O AUXÍLIO DA GERÊNCIA DE CONFIGURAÇÃO DE SOFTWARE

Produtos de *software* normalmente envolvem quantidades significativas de artefatos que são manuseados por diversas pessoas envolvidas em seu projeto. Com o auxílio da GCS é possível responder às seguintes questões básicas, que depois são desmembradas em outras questões mais específicas: (1) Quais modificações aconteceram no *software*? (2) Por que estas modificações aconteceram? (3) O sistema continua íntegro mesmo depois das modificações?

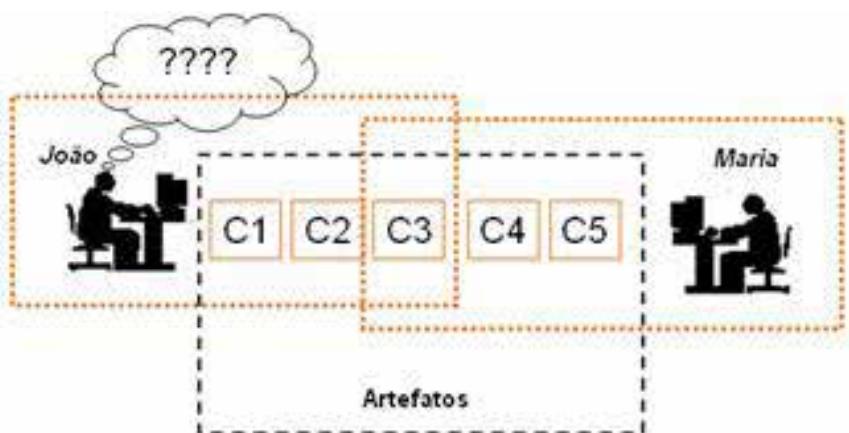
Sommerville (2011), define que a GCS está relacionada com as políticas, processos e ferramentas para gerenciamento de mudanças dos sistemas de *software*. É fácil perder o controle de quais mudanças e versões de componentes foram incorporadas em cada versão de sistema. As versões implementam propostas de mudanças, correções de defeitos e adaptações de *hardware* e sistemas operacionais diferentes. Pode haver várias versões em desenvolvimento e em uso ao mesmo tempo. Se não tiver GCS pode desperdiçar esforço modificando a versão errada de um sistema, entregá-la para o cliente ou esquecer onde está armazenado o código-fonte do *software* para uma versão específica do sistema ou componentes.



Cristine Dantas publicou um artigo sobre Gerência de Configuração em <<http://www.devmedia.com.br/gerencia-de-configuracao-de-software/9145>>, apresentando um cenário muito didático para explicar a importância da GCS. Veja a seguir seu exemplo.

Supomos que uma empresa de desenvolvimento de *software* não possui Gcs e que em um determinado projeto um programador esteja modificando os artefatos C1, C2 e C3 em um diretório compartilhado na rede. Em paralelo um outro programador modifica os artefatos C4, C5 e também o artefato C3, como exemplifica a figura abaixo.

FIGURA 50 – ESPAÇO DE TRABALHO COMPARTILHADO POR VÁRIOS DESENVOLVEDORES



FONTE: Disponível em: <<http://www.devmedia.com.br/gerencia-de-configuracao-de-software/9145>>. Acesso em: 23 ago. 2015.

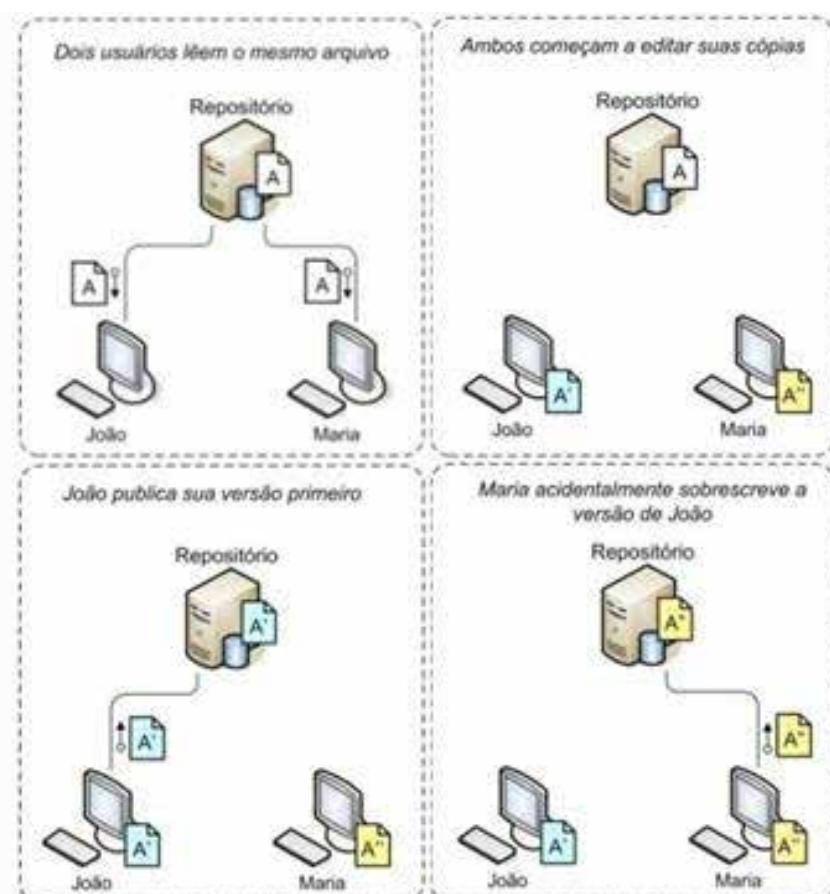
O programador João da figura acima, poderia implementar sua modificação em uma versão desatualizada do artefato e sobrepor a versão mais atual disponibilizada por Maria. Se João não notificar Maria sobre o impacto que a

modificação do artefato C3 pode causar no código, esta falta de compartilhamento faz com que o primeiro seja perdido. Maria não conseguirá identificar, de forma rápida, o motivo que levou sua implementação a falhar. Como também ocorre de os dois trabalharem em suas máquinas e após as modificações cada um devolver separadamente para o repositório principal.

Sem GCS não seria possível guardar e mesclar estas mudanças, e a última versão que subisse ao repositório principal eliminaria a outra parte implementada e adicionada, uma vez que, iria só comparar a segunda alteração com a versão original esquecendo-se de comparar com outras alterações e realizar o merge entre elas, sem sobreposição, mas junção.

Este problema ocorre devido à atualização simultânea, quando dois programadores compartilham o mesmo repositório e não existe controle ou restrição quanto ao acesso a este repositório (ver Figura abaixo).

FIGURA 51 – REPOSITÓRIO CENTRALIZADO COMPARTILHADO POR VÁRIOS DESENVOLVEDORES

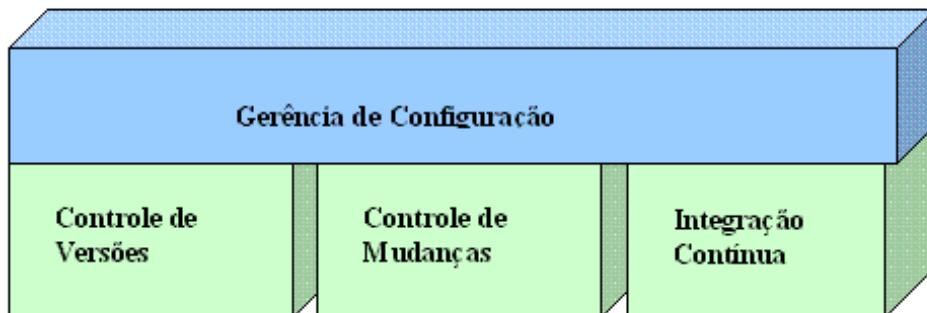


FONTE: Disponível em: <<http://www.devmedia.com.br/gerencia-de-configuracao-de-software/9145>>. Acesso em: 23 ago. 2015.

Por fim, Dantas (2015) destaca que sob a perspectiva de desenvolvimento, a GCS abrange três sistemas principais: controle de modificações, controle de versões e controle de gerenciamento de construção. E Roger Pressman (2011) nos explica em outros termos sobre estas três partes:

1. Controle de Versão: cuida das mudanças realizadas nos artefatos, salvando o artefato em um repositório que pode ser obtido (*check-out*) por algum desenvolvedor, modificado e depois, atualizado (*check-in*). Para cada atualização feita, é salvo uma nova versão do artefato, guardando também um histórico das versões e suas mudanças.
2. Controle de Mudanças: visa documentar as alterações que serão realizadas no sistema, como sistemas de controle de tarefas, podendo ser rastreado depois o que mudou, por que mudou, quem mudou etc.
3. Por fim, Integração Contínua: tem como objetivo montar um ambiente que junta a última versão de todos os artefatos de um *software*, logo quando são alterados, e os testa para garantir que as alterações estão consistentes.

FIGURA 52 – PONTOS FUNDAMENTAIS DA GC



FONTE: Disponível em: <<http://www.devmedia.com.br/quick-tips-gerencia-de-configuracao-parte-1/13708>>. Acesso em: 24 ago. 2015.

Algum mecanismo de controle é necessário para gerenciar a entrada e saída dos componentes a fim de evitar problema de atualização simultânea.

O gerenciamento da GCS não é algo tão fácil, porém para garantir a qualidade na aplicação de seus controles, revisões e auditorias de configuração de *software* são necessárias a fim de descobrir omissões ou erros na configuração e se os procedimentos, padrões, regulamentações ou guias foram devidamente aplicados no processo e no produto.

O último passo do processo de GCS é a preparação de relatórios, tarefa que tem como objetivo relatar a todas as pessoas envolvidas no desenvolvimento e manutenção do *software* o que aconteceu nos artefatos, quem fez, data da alteração, o que mais será afetado, etc. Portanto, o acesso rápido às informações agiliza o processo de desenvolvimento e melhora a comunicação entre as pessoas,

evitando, assim, muitos problemas de alterações do mesmo item de configuração, com intenções diferentes e, às vezes, conflitantes.

Pressman (1995) finaliza informando que o controle de versão combina procedimentos e ferramentas para gerenciar diferentes versões de objetos de configuração que são criadas durante o processo de Engenharia de *Software*.

4 AUTORIA DE CONFIGURAÇÃO

A identificação, controle de versão e o controle de mudanças ajudam o desenvolvedor de *software* a manter ordem naquilo que, de outro modo, seria uma situação caótica e fluídica. Porém, mesmo os mais bem-sucedidos mecanismos de controle rastreiam uma mudança somente até o ponto em que uma solicitação de mudança é gerada. Como podemos garantir que a mudança foi adequadamente implementada? A resposta é dupla: (1) revisões técnicas formais e (2) a auditoria de configuração de *software*.

A revisão técnica formal focaliza a exatidão técnica do objeto de configuração que foi modificado. Os revisores avaliam sua consistência com outros itens de configuração, omissões ou potenciais efeitos colaterais. Uma revisão técnica formal deve ser realizada para todas as mudanças mesmo para as mais triviais.

Uma auditoria de configuração de *software* complementa a revisão técnica formal ao avaliar um item de configuração quanto às características que geralmente não são consideradas durante a revisão. A auditoria pergunta e responde as seguintes questões:

1. A mudança especificada na solicitação foi feita? Outras modificações adicionais foram incorporadas?
2. Uma revisão técnica formal foi realizada para avaliar a exatidão técnica?
3. Os padrões de engenharia de *software* foram adequadamente seguidos?
4. A mudança foi “realçada” no item de configuração de *software*? A data de mudança e o autor de mudança foram especificados? Os artefatos do objeto de configuração refletem a mudança?
5. Os procedimentos de solicitações de mudança para anotar a mudança, registrá-la e relatá-la foram seguidos?
6. Todos os itens de configuração relacionados foram adequadamente atualizados?

Em alguns casos, as questões de auditoria são levantadas como parte de uma revisão técnica formal. Porém, quando a solicitação de mudança é uma atividade formal, a auditoria é realizada separadamente pelo grupo de garantia de qualidade.

5 RELATÓRIO DE STATUS DE CONFIGURAÇÃO

A produção do relatório de status de configuração é uma tarefa da solicitação de mudança que responde às seguintes perguntas: (1) O que aconteceu? (2) Quem o fez? (3) Quando aconteceu? (4) O que mais será afetado? Todas as vezes que uma solicitação de mudança recebe uma identificação nova ou atualizada, uma entrada no relatório de status de configuração é feita.

Todas as vezes que uma auditoria de configuração é realizada, os resultados são registrados como parte da tarefa de solicitação de mudanças. A saída do relatório de status de configuração pode ser colocada num banco de dados *on-line*, de forma que os desenvolvedores ou realizadores de manutenção de *software* possam acessar as informações sobre mudanças por meio de categorias de palavras-chave. Além disso, um relatório de status de configuração é gerado regularmente, e ele tem a intenção de manter a administração e os profissionais a par de mudanças importantes.

O relatório de status de configuração desempenha um papel vital no sucesso de um grande projeto de desenvolvimento de *software*. Quando muitas pessoas estão envolvidas, é provável que ocorra a situação de “a mão esquerda não saber o que a direita está fazendo”. Dois desenvolvedores podem tentar modificar a mesma solicitação de mudança com intenções diferentes e conflitantes.

Uma equipe de engenharia de *software* pode gastar meses de esforço construindo um *software* para uma especificação de *hardware* obsoleta. A pessoa que reconheceria efeitos colaterais sérios para uma mudança proposta não tem consciência de que a mudança está sendo feita. O relatório de status de configuração ajuda na eliminação desses problemas ao melhorar a comunicação entre as pessoas envolvidas.

FONTE: PRESSMAN, Roger S. **Engenharia de Software**. São Paulo: Makron Books, 1995.

6 TERMINOLOGIAS DA GERÊNCIA DE CONFIGURAÇÃO DE SOFTWARE

A seguir serão apresentadas algumas terminologias utilizadas pela área de GCS definidas por Ian Sommerville (2011) e Kevin Cabral (2015):

- **Baseline:** é uma base estável para a evolução contínua dos itens de configuração (IC), foi formalmente revisado e aceito e serve como marco para os próximos passos de desenvolvimento, onde um artefato ou vários só se torna um item de configuração depois que um *baseline* é estabelecido.

- **Repositório:** local físico e lógico onde os itens de um sistema são guardados. Pode conter diversas versões do sistema e utiliza mecanismos de controle de acesso.
- **Check-Out:** recupera a última versão de um IC guardada no repositório e as permissões aos usuários podem ser de dois tipos: (1) Escrita (Cria uma cópia, para edição no cliente) e (2) Leitura (Cria uma cópia, apenas para leitura no cliente).
- **Check-In:** ação de inserir/atualizar um IC no repositório: Verifica o IC, caso o mesmo já exista, Verifica e incrementa a versão do item, Registra informações das mudanças (autor, data, hora, comentários), Inclui/atualiza o item.
- **Build:** é uma versão "compilada" de um *software* ou parte dele que contém um conjunto de recursos que poderão integrar o produto final. O espaço para integração de funcionalidades, onde inclui não só código fonte, mas documentação, arquivos de configuração, base de dados etc.
- **Release:** identificação e empacotamento de artefatos entregues ao cliente (interno ou externo) ou ao mercado, onde um release implica no estabelecimento de um novo *baseline*, de produto. Produto de *software* supostamente sem erros. Versão do sistema validada após os diversos tipos de teste. Processo iterativo/incremental produz, em geral, mais de um release.
- **Branch:** criação de um fluxo alternativo para atualização de versões de itens de configuração. Devem existir regras bem definidas para criação de *branches*. Por que e quando devem ser criados? Quais os passos? Quando retornar ao fluxo principal?
- **Merge:** unificação de diferentes versões de um mesmo item de configuração. Integração dos itens de configuração de um *branch* com os itens de configuração do fluxo principal. *Check-out* atualizando a área local. Algumas ferramentas fornecem um mecanismo automático para realização de merges. Mesmo com o uso de ferramentas, em vários casos há necessidade de intervenção humana.

Existem diversas ferramentas disponíveis no mercado para apoiar a Gerência de Configuração de Sistemas em controle de versões. As funcionalidades, documentação, o suporte disponível e popularidade variam bastante. As ferramentas comerciais apresentam maior número de funcionalidades mas têm um alto custo. As ferramentas *open source* além de baixo custo, também apresentam vantagens como qualidade e segurança. O quadro abaixo apresenta algumas ferramentas e suas aplicações.

QUADRO 2 - FERRAMENTAS DE GCS

Controle de versão:	Controle de Mudanças:	Integração Contínua:
Open Source: Mercurial, Git, Subversion, CVS Comercial: Team Foundation Server (Microsoft), Team Concert (IBM/Rational), ClearCase, StarTeam, Perforce, BitKeeper.	Open Sources: Trac, Redmine, Mantis e Bugzilla. Comerciais: JIRA, CaliberRM, Perforce e FogBUGZ. Mantis, Redmine e Bugzilla.	Open Sources: Scons, Jenkins, Hudson, Maven, Gump e Ant. Comerciais: AntHill Pro, FinalBuilder e BuildForge. Jenkins Jenkins.

FONTE: O autor.

O alto número de ferramentas *Open Source* torna a implantação de algum tipo de GCS fácil e barata, principalmente para micro e pequenas empresas. O que ainda atrapalha um pouco, é o esforço necessário para a adequação do processo e os treinamentos específicos.

RESUMO DO TÓPICO 3

Neste tópico você aprendeu que:

- A Gerência de Configuração de *Software* (GCS) serve para controlar as alterações durante o desenvolvimento de *software* por meio de métodos e ferramentas a fim de minimizar os erros cometidos durante sua evolução.
- A GCS propõe automação, versionamento, gerência de mudanças e gerência de *releases*, mantendo a consistência e a integridade do *software* com as especificações.
- Itens de Configuração (IC) são todos os artefatos como os arquivos-fonte que compõem um produto de *software*.
- Referente às atualizações simultâneas de artefatos num determinado repositório, sem GCS não seria possível guardar e mesclar estas mudanças, e a última versão que subisse ao repositório principal eliminaria a outra parte implementada e adicionada, uma vez que iria só comparar a segunda alteração com a versão original esquecendo-se de comparar com outras alterações e realizar o merge entre elas, sem sobreposição, mas junção.
- A GCS abrange três sistemas principais: controle de modificações, controle de versões e controle de gerenciamento de construção.
 - ✓ Controle de Versão: para cada atualização feita, é salvo uma nova versão daquele artefato, guardando um histórico das versões e suas mudanças;
 - ✓ Controle de Mudanças: Visa documentar as alterações que serão realizadas no sistema, como sistemas de controle de tarefas, podendo ser rastreado depois o que mudou, por que mudou, quem mudou etc.;
 - ✓ Integração Contínua: tem como objetivo montar um ambiente que junta a última versão de todos os artefatos de um *software*, logo quando são alterados, e os testa para garantir que as alterações estão consistentes.
- O processo de GCS é preparação de auditorias e de relatórios, tarefa que tem como objetivo relatar a todas as pessoas envolvidas no desenvolvimento e manutenção do *software* o que aconteceu nos artefatos, quem fez, data da alteração, o que mais será afetado etc.
- A auditoria de configuração é uma atividade que ajuda a garantir que a qualidade seja mantida à medida que as mudanças são feitas. O relatório de status de configuração oferece informações sobre cada mudança àqueles que precisam tomar conhecimento delas.

- Terminologias utilizadas pela área de GCS são:
 - ✓ *Baseline* que é uma base estável para a evolução contínua dos itens de configuração (IC).
 - ✓ *Repositório*: Local físico e lógico onde os itens de um sistema são guardados.
 - ✓ *Check-Out*: Recupera a última versão de um IC guardada no repositório.
 - ✓ *Check-In*: Ação de inserir/atualizar um IC no repositório.
 - ✓ *Build*: É uma versão “compilada” de um *software* ou parte dele que contém um conjunto de recursos que poderão integrar o produto final.
 - ✓ *Release*: Identificação e empacotamento de artefatos entregues ao cliente.
 - ✓ *Branch*: Criação de um fluxo alternativo para atualização de versões de itens de configuração.
 - ✓ *Merge*: Unificação de diferentes versões de um mesmo item de configuração. Integração dos itens de configuração de um *branch* com os itens de configuração do fluxo principal.
- Existem diversas ferramentas para controle de versão, controle de mudanças e integração contínua disponíveis no mercado (*Open Source* ou Comerciais) para apoiar a Gerência de Configuração de *Software*, oferecendo suporte para facilitar a construção de diferentes versões de uma aplicação.

AUTOATIVIDADE



1 No processo de desenvolvimento de *software*, o gerenciamento da configuração de *software* envolve identificar a sua configuração:

- a) () antes do início do ciclo de vida
- b) () apenas no início do ciclo de vida.
- c) () somente ao final do ciclo de vida.
- d) () apenas uma vez antes de se encerrar o ciclo de vida.
- e) () em pontos predefinidos no tempo durante o ciclo de vida.

2 Um *software* de gerência de configuração deve ser capaz de:

- a) () configurar o *software* de acordo com as preferências do usuário.
- b) () gerenciar que usuários podem ter acesso a certos tipos de funções do *software*.
- c) () atualizar automaticamente as versões do *software* instaladas nas máquinas clientes.
- d) () gerenciar a evolução do *software* durante o seu processo de desenvolvimento.
- e) () configurar automaticamente a instalação do *software* de acordo com o sistema operacional do servidor.

3 No que diz respeito à área da engenharia de *software*, analise a citação a seguir.

“Conjunto de atividades projetadas para controlar as mudanças pela identificação dos produtos do trabalho que serão alterados, estabelecendo um relacionamento entre eles, definindo o mecanismo para o gerenciamento de diferentes versões destes produtos, controlando as mudanças impostas, e auditando e relatando as mudanças realizadas.”

Essa citação apresenta o conceito de:

- a) () Auditoria de Configuração.
- b) () Gestão de Configuração.
- c) () Gerência de Mudanças.
- d) () Controle de Versão.
- e) () Versões de Projeto.

- 4 No processo de gerenciamento de configuração de *software*, um conjunto de itens de configuração que deve ser controlado, formalmente designado e fixado num tempo específico do ciclo de vida do *software*, é denominado.
- () configuração de *software*.
 - () relações de itens.
 - () aquisição de itens.
 - () *baseline*.
 - () versão.

- 5 Gerência de Configuração de *Software* é um conjunto de atividades de apoio, que permite a absorção controlada das mudanças, inerentes ao desenvolvimento de *software*, mantendo estabilidade na evolução do projeto. Podemos dividir o Gerenciamento de Configuração de *Software* em três níveis:

Gerência de Configuração		
Controle de Versão	Controle de Mudanças	Integração Contínua

Tomando por base o quadro anterior, assinale a alternativa que define, corretamente, cada nível do Gerenciamento de Configuração de *Software*.

- () Controle de versão: identifica em qual versão está o *software* e quais as características de cada versão. Controle de mudanças: identificam quais foram as mudanças efetuadas na versão. Integração contínua: tem como características testar as mudanças, assim que são realizadas.
- () Controle de versão: identifica em qual versão está o *software* e quais as características de cada versão. Controle de mudanças: tem como característica testar as mudanças, assim que são realizadas. Integração contínua: identificam quais foram as mudanças efetuadas na versão.
- () Controle de versão: identifica quais foram as mudanças, efetuadas na versão. Controle de mudanças: tem como característica testar as mudanças, assim que são realizadas. Integração contínua: identifica em qual versão está o *software* e quais as características de cada versão.
- () Controle de versão: identifica os usuários dos sistemas. Controle de mudanças: tem como característica testar as mudanças, assim que são realizadas. Integração contínua: identifica em qual versão está o *software* e quais as características de cada versão.
- () Controle de versão: identifica os usuários dos sistemas. Controle de mudanças: tem como característica testar as mudanças, assim que são realizadas. Integração contínua: identifica as aplicações do *software*.

6 Sistemas de controles de versões são ferramentas essenciais na gestão de tecnologia da informação de empresas, em especial em empresas desenvolvedoras de *software*. Estes sistemas têm o intuito de:

- a) () Alocar recursos específicos para o desenvolvimento de diferentes versões do sistema.
- b) () Calcular as funcionalidades do sistema, incluindo cálculos de pontos de função.
- c) () Identificar uma alteração específica efetuada em um código fonte.
- d) () Controlar as versões dos diversos *softwares* adquiridos pela empresa.
- e) () Estimar o custo e tempo de desenvolvimento de uma versão específica de um sistema.

GERENCIAMENTO DE QUALIDADE DE SOFTWARE: PADRÕES, NORMAS E MODELOS; MÉTODOS ÁGEIS; VERIFICAÇÃO, VALIDAÇÃO E TESTES DE SOFTWARE; GOVERNANÇA DE TECNOLOGIA DA INFORMAÇÃO

OBJETIVOS DE APRENDIZAGEM

Ao final desta unidade, você será capaz de:

- entender os conceitos sobre os principais padrões, normas e modelos de Qualidade de Software;
- entender a importância dos Métodos Ágeis e suas principais metodologias de desenvolvimento de Softwares;
- compreender as áreas de Testes Software;
- conhecer a importância da Governança de TI nas organizações.

PLANO DE ESTUDOS

Esta unidade de ensino contém quatro tópicos, sendo que no final de cada um, você encontrará atividades que contribuirão para a apropriação dos conteúdos.

TÓPICO 1 – GERENCIAMENTO DE QUALIDADE DE SOFTWARE: PADRÕES, NORMAS E MODELOS

TÓPICO 2 – MÉTODOS ÁGEIS

TÓPICO 3 – VERIFICAÇÃO, VALIDAÇÃO E TESTES DE SOFTWARE

TÓPICO 4 – GOVERNANÇA DE TECNOLOGIA DA INFORMAÇÃO

GERENCIAMENTO DE QUALIDADE DE SOFTWARE: PADRÕES, NORMAS E MODELOS

1 INTRODUÇÃO

O termo Qualidade, dependendo do ponto de vista e do grau de importância, está relacionado a uma série de aspectos, algo difícil de ser definido e ainda mais difícil de ser garantido em qualquer necessidade. No caso de um curso de graduação *on-line*, fatores como matriz curricular, material didático e formação acadêmica dos professores têm estreita relação com a qualidade.

Pensa-se em qualidade como algo dispendioso e caro de se obter, pois nos parece lógico que uma alta qualidade requer mais testes, inspeções, ajustes e cuidados extras durante a implementação e entrega de um serviço ou produto de *software*. Qualidade é a efetividade dos nossos produtos e serviços em satisfazer os nossos clientes. É “O que” nós fornecemos ao cliente, julgado certo ou errado somente pelo cliente, baseado nas suas expectativas e percepções.

O objetivo principal da gerência de qualidade é obter assertividade e produtividade durante a execução de nossas atividades. Portanto, produtividade pode ser definida com a eficiência com a qual se prove a qualidade requerida pelos clientes. É o “Como” fornecemos serviços e produtos de qualidade.

Na área de prestação de serviços, produtividade (rapidez, eficiência) é usualmente um dos principais critérios de qualidade definido pelo cliente: portanto um prestador de serviços deve considerar “Produtividade” como um elemento ou aspecto principal da “qualidade”.

Em desenvolvimento de *software* a qualidade deve ser entendida nos aspectos da correta compreensão dos requisitos do cliente, quando se desenvolve o projeto com zero defeito, quando se obtém aumento de produtividade e redução de custos e, por fim, uma boa usabilidade do Sistema. A qualidade está fortemente relacionada à conformidade com os requisitos, ou seja, atender ao que o usuário pede formalmente. Na área de Engenharia de *Software*, Roger Pressman (2011) define qualidade como “Conformidade a requisitos funcionais e de desempenho explicitamente declarados, a padrões de desenvolvimento claramente documentados e a características implícitas que são esperadas de todo *software* profissionalmente desenvolvido”.

A Organização Internacional de Padronização (*International Organization for Standardization - ISO*) através da ISO 9000 define qualidade como a totalidade das características de uma entidade que lhe confere a capacidade de satisfazer

às necessidades explícitas e implícitas. Necessidades explícitas são as condições e objetivos propostos por aqueles que produzem o *software*. As necessidades implícitas são necessidades subjetivas dos usuários, também chamadas de fatores externos, e podem ser percebidas tanto pelos desenvolvedores quanto pelos usuários (ISO, 2015).

2 ABORDAGENS DA QUALIDADE

Para entender melhor a abordagem da qualidade implantada há quase 70 anos nas indústrias seguem algumas das principais características conforme alguns idealizadores da qualidade (VASCONCELOS, 2006):

- Armand Feigenbaum era estudante de doutorado no *Massachusetts Institute of Technology* nos anos 1950, quando introduziu o conceito de Controle da Qualidade Total que representa um sistema efetivo que integra a qualidade do desenvolvimento, qualidade de manutenção e esforços de melhoria da qualidade de vários grupos em uma organização.
- Joseph M. Juran foi um educador chave em gerência da qualidade para os japoneses, que difundiram sua teoria. Ele fundamentou sua abordagem em três processos básicos: **Planejamento da Qualidade**: ser iniciado com a determinação das necessidades do cliente, seguido da definição de requisitos básicos para que o produto ou serviço atenda às expectativas do cliente. **Controle da Qualidade**: uma vez estabelecidos direcionamentos e processos para desenvolvimento dos produtos e serviços, garante o grau de qualidade e produtividade previamente estabelecido. **Melhoria da Qualidade**: a identificação de oportunidades de melhoria, assim como o estabelecimento de responsabilidades para execução dessas melhorias e divulgação de resultados.
- Philip B. Crosby iniciou seus estudos na década de 1960 e defendeu que a qualidade pode ser vista em quatro “certezas” do gerenciamento da qualidade: qualidade significa atendimento aos requisitos; qualidade vem através de prevenção; padrão para desempenho da qualidade e “defeitos zero”; e a medida de qualidade é o preço da não conformidade.



Os principais gurus da Qualidade Mundial são entre outros: Armand Feigenbaum, William Edwards Deming, Joseph M. Juran, Karou Ishikawa e Tom Peters. A seguir apresento um texto completo dos pesquisadores mais reconhecidos na área: <<http://falandodequalidade.net/Gurus%20da%20Qualidade%20Mundial%20PageView.pdf>>. Boa leitura!

Uma das principais formas de implementação do controle de qualidade é a utilização do Ciclo PDCA (*Plan-Do-Check-Action*), que consiste em quatro fases: planejar, fazer, checar e agir corretamente. O PDCA deve ser utilizado para todas as organizações na definição de uma metodologia de controle ou melhoria de qualquer tipo de processo.

FIGURA 53 – CICLO PDCA PARA MELHORIA



FONTE: Disponível em: <<https://blogpegg.files.wordpress.com/2011/03/ciclo-pdca.jpg>>. Acesso em: 20 set. 2015.

Na fase *Plan*, o foco está na identificação do problema, análise do processo atual e definição do plano de ação para melhoria do processo em questão. Na fase *Do*, o plano de ação definido deve ser executado e controlado. Em *Check*, verificações devem ser realizadas, a fim de subsidiar ajustes e se tirar lições de aprendizagem. Finalmente, em *Action*, deve-se atuar corretivamente para fundamentar um novo ciclo, garantindo a melhoria contínua (VASCONCELOS et al., 2006).

A partir de 1947 a ISO formalizou a necessidade da definição de padrões e normas internacionais que contribuam para um PDCA direcionado a gestão da qualidade aplicada para produção, serviços e gerenciamento de um produto, desde seu pedido, passando pela análise e gerenciamento de requisitos, fabricação até entrega ao cliente, incluindo infraestrutura adequada, competências e comprometimento da alta administração.

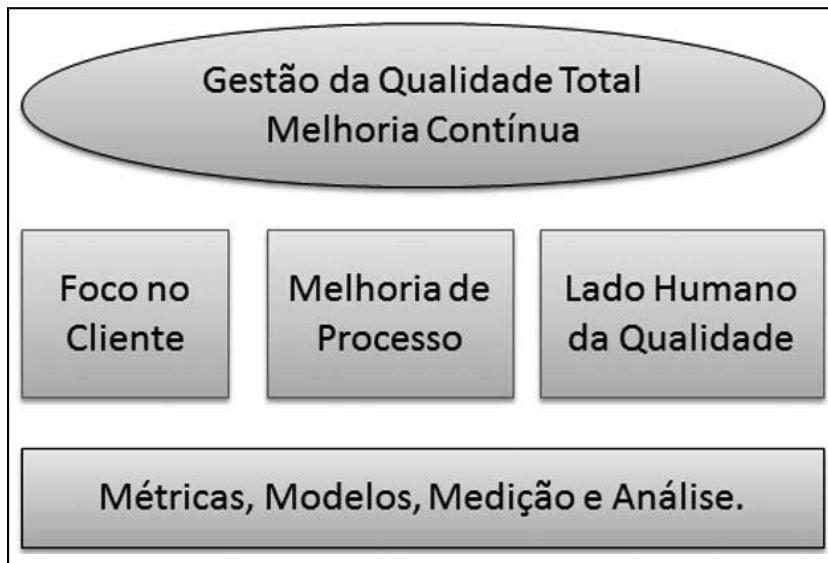
3 TOTAL QUALITY MANAGEMENT (TQM)

O gerenciamento da qualidade de *software* teve origem no *Total Quality Management* (TQM) à medida que as organizações começaram a buscar na sua cultura aplicar a melhoria de processos, produtos e serviços a fim de obter maior eficácia, eficiência e satisfação organizacional.

Kan (1995) descreve os seguintes elementos-chave do TQM, conforme demonstrado graficamente na figura a seguir:

- **Foco no cliente:** consiste em analisar os desejos e necessidades, bem como definir os requisitos do cliente, além de medir e gerenciar a satisfação do cliente;
- **Melhoria de processo:** o objetivo é reduzir a variação do processo e atingir a melhoria contínua do processo. Processos incluem tanto processos de negócio quanto processo de desenvolvimento de produtos;
- **Aspecto humano:** nesse contexto, as áreas-chave incluem liderança, gerência, compromisso, participação total e outros fatores sociais, psicológicos e humanos;
- **Medição e análise:** o objetivo é gerenciar a melhoria contínua em todos os parâmetros de qualidade por um sistema de medição orientado a metas.

FIGURA 54 – ELEMENTOS-CHAVE DO TQM



FONTE: Adaptado de: Kan (1995)

O TQM dividiu o desenvolvimento de *software* em dois momentos: a fase do *software* artesanal e a do *software* profissional. Na fase artesanal, antes da definição de Engenharia de Software em 1969, o *software* era em geral desenvolvido e testado continuamente até se chegar a algo que funcionasse e que o cliente pudesse

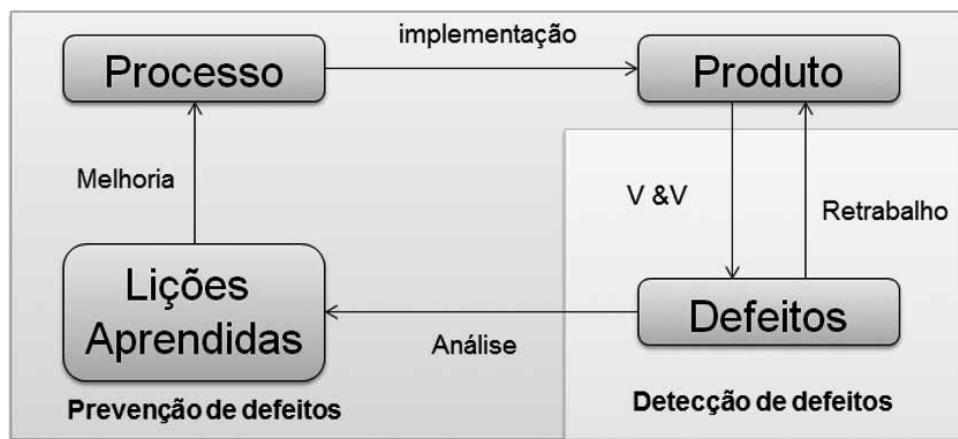
aceitar. Na fase profissional, já a era da Engenharia de *software*, além de funcionar e ser aceito pelo cliente, o *software* precisava ser padronizado, documentado e com boa relação custo/benefício (HIRAMA, 2012).

4 INTRODUÇÃO À QUALIDADE DE SOFTWARE

Qualidade de *software* está relacionada a entregar ao cliente o produto final que satisfaça suas expectativas, dentro daquilo que foi acordado inicialmente por meio dos requisitos do projeto. Nesse contexto, qualidade de *software* que objetiva garantir essa qualidade pela definição de processos de desenvolvimento (ENGHOLM JR., 2010).

Para produzir um produto de *software* com qualidade deve-se possuir processos formais que visem à prevenção e detecção de defeitos durante o desenvolvimento de *software*. A origem do produto se dá pela implementação de um processo consistente e em constante melhoria contínua.

FIGURA 55 – DESENVOLVIMENTO DE PRODUTO DE SOFTWARE



FONTE: O autor

Várias técnicas são utilizadas para identificar defeitos nos produtos de trabalho. Esses defeitos são eliminados através de retrabalho, que têm efeito imediato na produtividade do projeto. Defeitos também são encontrados em atividades de teste e podem ser analisados, a fim de se identificar suas causas. A partir dessa análise, lições aprendidas podem ser usadas para criar futuros produtos e prevenir futuros defeitos e, dessa forma, ter impacto positivo na qualidade do produto e na produtividade do projeto.

FONTE: Disponível em: <http://www.infoclad.com.br/apostilas/Engenharia%20de%20Software/Apostila%20de%20Engenharia%20de%20Software/Cap%EDtulos/UNIDADE%20II/CAP%CDTULO%2015%20DA%20APOSTILA_QUALIDADE%20DE%20SOFTWARE_2%AA%20%20PARTE.doc>. Acesso em: 8 out. 2015.

Todo processo de *software* deve possuir junto ao plano de projeto uma documentação específica da qualidade, denominada como plano de qualidade que deve compreender informações sobre como a equipe de qualidade irá garantir o cumprimento da política de qualidade, no contexto do programa ou projeto a serem desenvolvidos, quais métodos, técnicas, métricas, treinamentos e padrões devam ser utilizados ao longo de todo o ciclo de vida do projeto. O plano deve oferecer a base do gerenciamento dos riscos, dos testes, das inspeções, das auditorias e como deverão ocorrer os reportes de problemas e ações corretivas.

Pode-se citar entre tantos outros exemplos, que a técnica de prevenção de defeitos em um processo de desenvolvimento de *software* se dá pelo uso de instruções de procedimentos (padrões formais), treinamentos, documentação, modelagem e reengenharia, já as técnicas de detecção de defeitos podem ser pela análise de código; revisão por pares; testes, auditorias, verificações e validações.

4.1 GARANTIA E CONTROLE DA QUALIDADE DE SOFTWARE

Na gestão da qualidade de *software* existem diversas atividades voltadas à garantia da qualidade e ao controle de qualidade de *software*. A primeira é para a definição padronizada das atividades voltadas a prevenção de defeitos e problemas, que podem surgir nos produtos de trabalho. Área que define padrões, metodologias, técnicas e ferramentas de apoio ao desenvolvimento tendo como entrada o plano de qualidade de *software* e os resultados de medições de qualidade. A segunda é voltada para o monitoramento de resultados específicos do projeto, ou seja, a detecção de defeitos, executadas através do uso de técnicas que incluem revisões por pares, teste e análise de tendências, entre outras. Abaixo consta a tabela com as principais diferenças entre elas (VASCONCELOS et al., 2006).

TABELA 14 – GARANTIA DE QUALIDADE X CONTROLE DE QUALIDADE

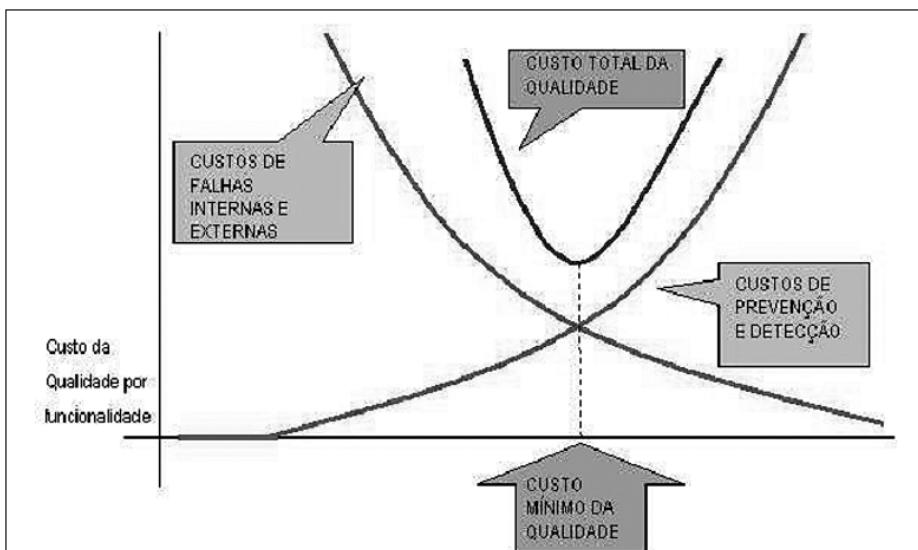
Garantia da Qualidade	Controle da Qualidade
<ol style="list-style-type: none">1. Garantia da qualidade garante que o processo é definido e apropriado.2. Metodologia e padrões de desenvolvimento são exemplos de garantia da qualidade.3. Garantia da qualidade é orientada a processo.4. Garantia da qualidade é orientada a prevenção.5. Foco em monitoração e melhoria de processo.	<ol style="list-style-type: none">1. As atividades de controle da qualidade focam na descoberta de defeitos em si específicos.2. Um exemplo de controle da qualidade poderia ser: "Os requisitos definidos são os requisitos certos?".3. Controle da qualidade é orientado a produto.4. Controle da qualidade é orientado a detecção.

- | | |
|---|---|
| <p>6. As atividades são focadas no início das fases no ciclo de vida de desenvolvimento de <i>software</i>.</p> <p>7. Garantia da qualidade garante que você está fazendo certo as coisas e da maneira correta.</p> | <p>5. Inspeções e garantia de que o produto de trabalho atenda aos requisitos especificados.</p> <p>6. As atividades são focadas no final das fases no ciclo de vida de desenvolvimento de <i>software</i>.</p> <p>7. Controle da qualidade garante que os resultados do seu trabalho são os esperados conforme requisitos.</p> |
|---|---|

FONTE: Disponível em: <<http://slideplayer.com.br/slide/3692245/>>. Acesso em: 20 set. 2015.

Para Kerzner (1998), o custo de qualidade é categorizado em custos de prevenção, custos de avaliação, custos de falhas internas e custos de falhas externas a fim de despender mais esforço em prevenção e detecção para assim reduzir defeitos. Conforme ilustrado na figura a seguir, existe um ponto ótimo de equilíbrio entre os diferentes custos de qualidade, que aponta um custo ideal total da qualidade. Vale salientar que esse ponto ótimo, no entanto, pode ser difícil de ser mensurado, considerando custos subjetivos de falhas externas, como a insatisfação do cliente, por exemplo.

FIGURA 56 – BALANCEAMENTO DO CUSTO DA QUALIDADE

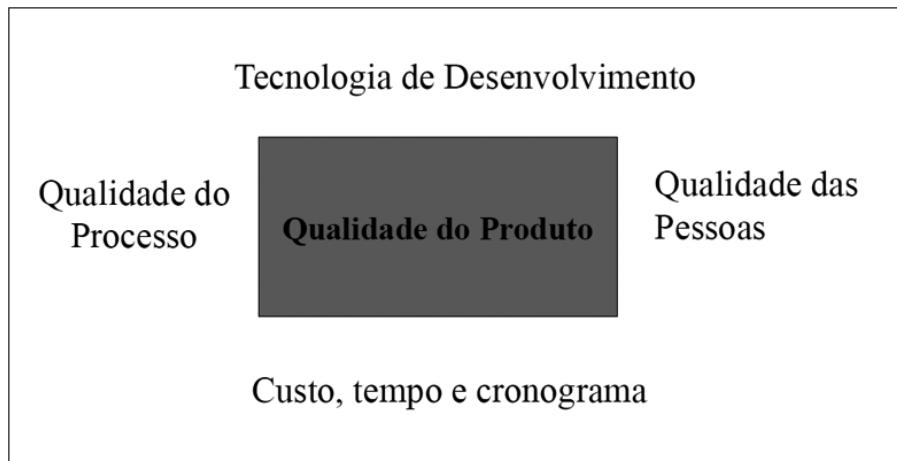


FONTE: Harold Kerzner (1998)

Para se obter um produto com qualidade, uma série de fatores são necessários: a qualidade do processo para um bom planejamento do projeto, a tecnologia de desenvolvimento utilizada através de ferramentas para construir o *software*, custo, tempo e cronograma implicam nos recursos financeiros e tempo

que terá para fazer o *software*, sendo estes os principais limitadores devido a recursos financeiros insuficientes e prazos muitos curtos. Por último tem-se a qualidade das pessoas, que é o fator decisivo, uma equipe bem qualificada realiza um trabalho de qualidade.

FIGURA 57 – PILARES DA QUALIDADE DE SOFTWARE



FONTE: Disponível em: <<http://image.slidesharecdn.com/aula14-1229053203557802-1/95/gerenciamento-de-qualidade-40-728.jpg?cb=1229024582>>. Acesso em: 19 set. 2015.

5 PADRÕES, NORMAS E MODELOS DE QUALIDADE DE SOFTWARE

Entre os principais objetivos da qualidade de *software* está a definição de técnicas e ferramentas para serem utilizadas durante o ciclo de vida do projeto, **PADRONIZANDO** a forma de realizar as atividades, um guia de trabalho proporcionando assertividade no projeto evitando erros humanos.

Diversos padrões e normas de qualidade de *software* vêm sendo propostos ao longo dos anos. Essas normas têm sido fortemente adotadas nos processos de *software* das organizações em todo o mundo. As normas da *International Organization for Standardization* (ISO, 2015) são padrões internacionais que “especificam requisitos para um sistema gerencial de qualidade de uma organização”.

Com o crescimento substancial das indústrias de *software* e, levando-se em conta que a produção de *software* apresenta características peculiares, a ISO tem trabalhado na definição de várias normas que podem ser utilizadas como guias e padrões para diversas áreas de atuação dentro do contexto da ciência da computação. A tabela a seguir apresenta algumas normas ISO aplicadas à qualidade de *software*, focadas em produto ou processo de *software*.

TABELA 15 – NORMAS ISO PARA SUPORTE AO DESENVOLVIMENTO DE SOFTWARE

Nome	Descrição
Norma ISO/IEC 9000	Apresenta diretrizes para a aplicação da ISO 9001 por organizações que desenvolvem <i>software</i> ao desenvolvimento, fornecimento e manutenção de <i>software</i> .
Norma ISO/IEC 12207	Define um processo de ciclo de vida de <i>Software</i> .
Norma ISO/IEC 15504	Focada na avaliação de processos organizacionais.
Norma ISO/IEC 9126	Define as características de qualidade de produto de <i>Software</i> (Funcionalidade, confiabilidade, eficiência, usabilidade, manutenibilidade e portabilidade).
Norma ISO/IEC 27000	Define um processo da Segurança da Informação.
Norma ISO/IEC 15939	Define um processo de Métrica de <i>Software</i> .

Fonte: O autor

A seguir serão apresentados estes principais padrões e normas internacionais utilizadas como suporte ao desenvolvimento de *software* nas organizações.

5.1 NORMA ISO/IEC 9000

A série de padrões ISO 9000 é um conjunto de documentos que engloba pontos referentes à garantia da qualidade em projeto, desenvolvimento, produção, instalação e serviços associados; objetivando a satisfação do cliente pela prevenção de não conformidades em todos os estágios envolvidos no ciclo da qualidade da empresa (BARDINE, 2015).

Segundo Portella (2015), a ISO9000 compreende um conjunto de cinco normas da ISO 9000 a ISO 9004:

- ISO 9000: Normas de gestão da qualidade e de garantia da qualidade (definir norma mais adequada);
- ISO 9001: Modelo de garantia de qualidade em projeto, instalação, desenvolvimento, produção e assistência técnica (ampla);
- ISO 9002: Modelo de qualidade em produção e instalação (não tem atividade de desenvolvimento);
- ISO 9003: Modelo para garantia da qualidade em inspeção e ensaios finais (testes);
- ISO 9004: Gestão da qualidade e elementos do sistema da qualidade (diretrizes).

Esta norma considera uma empresa como uma rede de processos interconectados e para um sistema de qualidade estar de acordo com a ISO, esses processos devem endereçar as áreas identificadas no padrão e devem ser documentados e praticados como desejado. Foram desenvolvidos para prover orientações para sua aplicação na especificação, desenvolvimento, instalação e suporte de *softwares*.



A ISO 9000 e suas normas são as mais utilizadas nas organizações do mundo todo. Como complemento aos estudos, favor acessar este endereço: <<http://www.coladaweb.com/administracao/iso-9000>>. Boa leitura!

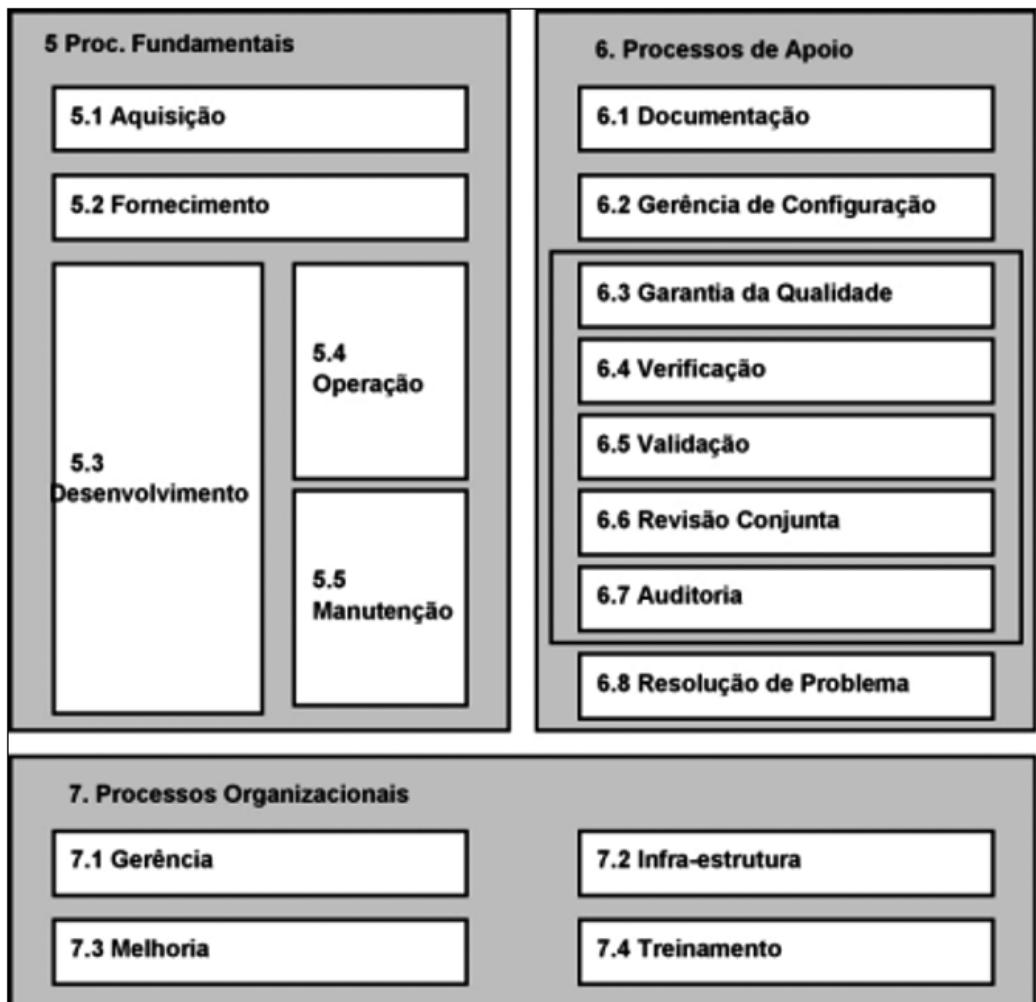
5.2 NORMA ISO/IEC 12207

O principal objetivo da norma ISO 12207 é o estabelecimento de uma estrutura comum utilizada como referência para os processos de ciclo de vida de *software* considerando que o desenvolvimento e a manutenção de *software* devem ser conduzidos da mesma forma que a disciplina de engenharia (ISO/IEC 12207:2008, 2015).

A estrutura descrita na ISO 12207 utiliza-se de uma terminologia bem definida e é composta de processos, atividades e tarefas a serem aplicados em operações que envolvam, de alguma forma, o *software*, seja através de aquisição, fornecimento, desenvolvimento, operação ou manutenção. Essa estrutura permite estabelecer ligações claras com o ambiente de engenharia de sistemas, ou seja, aquele que inclui práticas de *software*, *hardware*, pessoal e negócios (LAHOZ, 2015).

Segundo Batebyte (2015), a estrutura da norma foi concebida de maneira a ser flexível, modular e adaptável às necessidades de quem a utiliza. Para isso, está fundamentada em dois princípios básicos: modularidade e responsabilidade. Modularidade, no sentido de processos com mínimo acoplamento e máxima coesão. Responsabilidade, no sentido de estabelecer um responsável único por cada processo, facilitando a aplicação da norma em projetos, onde várias pessoas podem estar legalmente envolvidas.

FIGURA 58 – PROCESSO DA ISO 12207



FONTE: Disponível em: <<http://www.batebyte.pr.gov.br/modules/conteudo/conteudo.php?conteudo=325>>. Acessado em: 24 set. 2015.

A ISO 12207 pode ser utilizada com qualquer modelo de ciclo de vida, método ou técnica de engenharia de *software* e linguagem de programação. Ela descreve uma série de atividades importantes para o desenvolvimento de *software*. São elas: Levantamento de Requisitos; Análise dos Requisitos do *Software*; Projeto da Arquitetura do *Software*; Projeto Detalhado do *Software*; Implementação; Codificação e testes; Integração; Teste de qualificação e Instalação (ISO/IEC 12207:2008, 2015).

5.3 NORMA ISO/IEC 15504

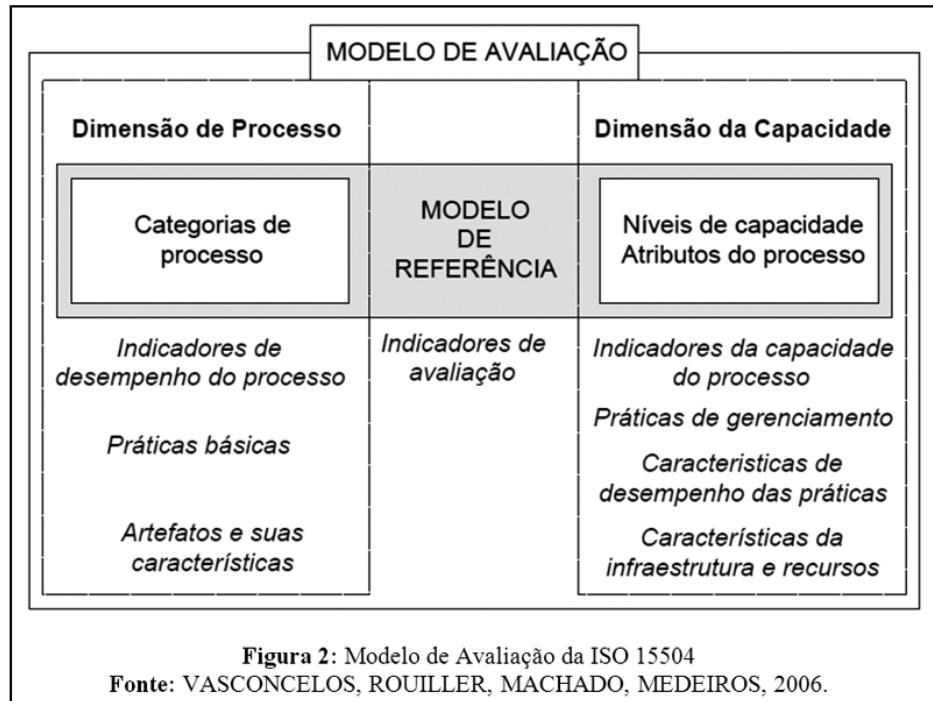
A ISO/IEC 15504 organiza e classifica as melhores práticas em duas dimensões: categorias de processo e níveis de capacidade. Cada uma das categorias de processo é detalhada em processos mais específicos, ou subcategorias (cliente-

fornecedor, engenharia, projeto, suporte e organização) (LAHOZ; SANT'ANNA, 2015).

Ainda segundo o mesmo autor, o modelo visa avaliar a capacidade da organização em cada processo, permitindo assim uma melhoria contínua. Cada um dos processos deve ser classificado em níveis (incompleto, executado, gerenciado, estabelecido, previsível e otimizado).

O ISO15504 possui um conjunto de nove documentos que endereçam avaliação de processo, assessoria de treinamento e competência, determinação da capacidade e melhoria de processo e está se tornando um modelo de referência para outros padrões como o CMMI, patrocinado pelo Departamento de Defesa norte americano, com colaborações da indústria, governo e pelo SEI. O CMMI tem como um de seus objetivos básicos, integrar os diversos modelos CMM existentes, bem como pretende garantir compatibilidade com a ISO15504, através da sua visão contínua de modelo, composto por áreas de processos universais e fundamentais. Possui ainda uma outra dimensão composta por uma métrica para a avaliação da capacidade de cada processo em uma organização (LAHOZ; SANT'ANNA, 2015).

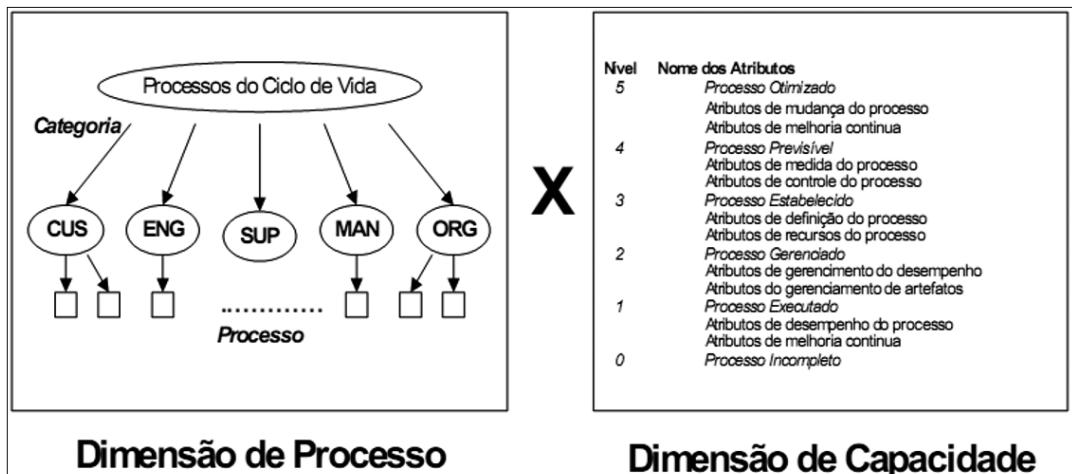
FIGURA 59 – MODELO DE AVALIAÇÃO



FONTE: Oliveira e Lima (2012)

A ISO/IEC 15504, também conhecida como SPICE, é a norma ISO/IEC que define processo de desenvolvimento de software. Ela é uma evolução da ISO/IEC 12207, mas possui níveis de capacidade para cada processo assim como o CMMI. Este modelo de referência possui uma abordagem bidimensional:

FIGURA 60 – AS DIMENSÕES DO MODELO DE REFERÊNCIA DA ISO 15504



FONTE: Disponível em: <http://blog.newtonpaiva.br/pos/wp-content/uploads/2012/06/e5i-19_ci-01.jpg>. Acessado em: 18 set. 2015.

A primeira dimensão auxilia os engenheiros de processo na definição dos processos necessários em uma organização de desenvolvimento de *software*. A segunda dimensão tem por objetivo determinar a capacidade do processo, avaliando-o através de um conjunto de atributos pré-estabelecidos.

Segundo a norma, uma avaliação de processo de *software* é uma investigação e análise disciplinada de processos selecionados de uma unidade organizacional em relação a um modelo de avaliação de processo.

Conforme {TR}Sampaio (2014), o processo de desenvolvimento de *software* definido pela ISO 15504 é dividido em três categorias principais de processos:

- Processos Primários contempla os processos de Aquisição, Fornecimento, Elicitação de Requisitos, Operação e categoria de Engenharia de *Software*.
- Processos Organizacionais contempla os processos de Gestão da qualidade; Gestão de riscos; Medição; Processos de Melhoria de Processos, Processos de Recursos e Infraestrutura e Processos de Reúso.

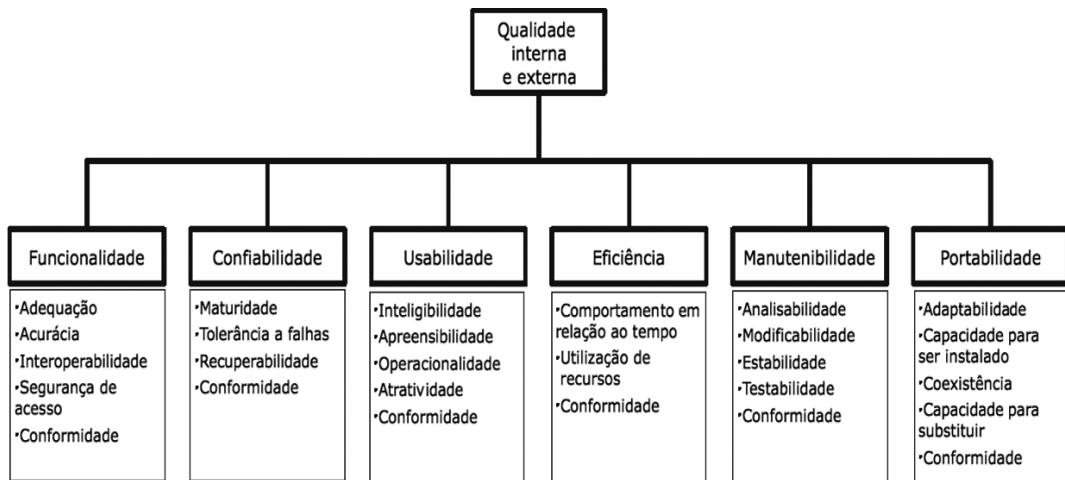
5.4 NORMA ISO/IEC ISO 9126

A ISO 9126 estabelece um modelo de qualidade para o produto de *software* que são avaliados conforme seis categorias básicas que são subdivididas em algumas características que são importantes para cada categoria (VENEZIANI, 2015):

- Funcionalidade: Capacidade de o *software* atender às necessidades do cliente.
- Confiabilidade: Segurança que o cliente tem para utilizar o *software*.
- Usabilidade: Facilidade do uso e entendimento das funcionalidades do sistema.

- Eficiência: Tempo necessário para a execução das funcionalidades dentro do “padrão”.
- Manutenibilidade: Facilidade para dar manutenção quando necessário.
- Portabilidade: Capacidade de adaptação do *software* para diversas plataformas.

FIGURA 61 – CATEGORIAS DA ISO 9126



FONTE: Disponível em: <<https://luizladeira.files.wordpress.com/2012/09/iso.jpg>>. Acesso em: 24 set. 2015.

A norma ISO/IEC 9126, ou conjunto de normas que trata deste assunto no âmbito da ISO, estabelece um modelo de qualidade com os seguintes componentes (ISO/IEC 9126, 2004):

- Processo de desenvolvimento, cuja qualidade afeta a qualidade do produto de *software* gerado e é influenciado pela natureza do produto desenvolvido;
- Produto, compreendendo os atributos de qualidade do produto (sistema) de *software*. Estes atributos de qualidade podem ser divididos entre atributos internos e externos. Estes se diferenciam pela forma como são aferidos (interna ou externamente ao produto de *software*) e em conjunto compõem a qualidade do produto de *software* em si;
- Qualidade em uso, que consiste na aferição da qualidade do *software* em cada contexto específico de usuário. Esta é, também, a qualidade percebida pelo usuário.



A parte 01 da ISO 9126 pode ser usada para especificar requisitos funcionais e não-funcionais do cliente e do usuário. Para maiores informações segue link: <http://luizcamargo.com.br/arquivos/NBR%20ISO_IEC%209126-1.pdf>. Boa leitura!

5.5 NORMA ISO/IEC 27000

A ISO 27000 trata sobre a área de segurança da informação e possui as seguintes famílias:

- ISO 27001 – Requisitos do SGSI;
- ISO 27002 – Controles de Segurança;
- ISO 27003 – Diretrizes de Implementação;
- ISO 27004 – Medição;
- ISO 27005 – Gestão de Risco;
- ISO 27006 – Auditoria de Segurança.

Geralmente, quando se fala em Segurança da Informação, o tema é associado a *hackers* e vulnerabilidades em sistemas, onde o principal entendimento é de que a empresa precisa de um bom antivírus, um *firewall* e ter todos os seus “*patches*” aplicados no ambiente tecnológico. Não há dúvida de que são questões importantes, porém a Segurança da Informação não está limitada a somente esses pontos.

É de praxe que as empresas protejam seus ativos de valor, tenham responsabilidade social e pratiquem a boa governança. Muitas organizações são obrigadas a cumprir legislação específica ou seguir regras definitivas para seu grupo de negócio. Assim, se faz necessário implementar e gerenciar práticas de Segurança da Informação. A série ISO/IEC27000 estabelece as normas para os Sistemas de Gerenciamento de Segurança da Informação:

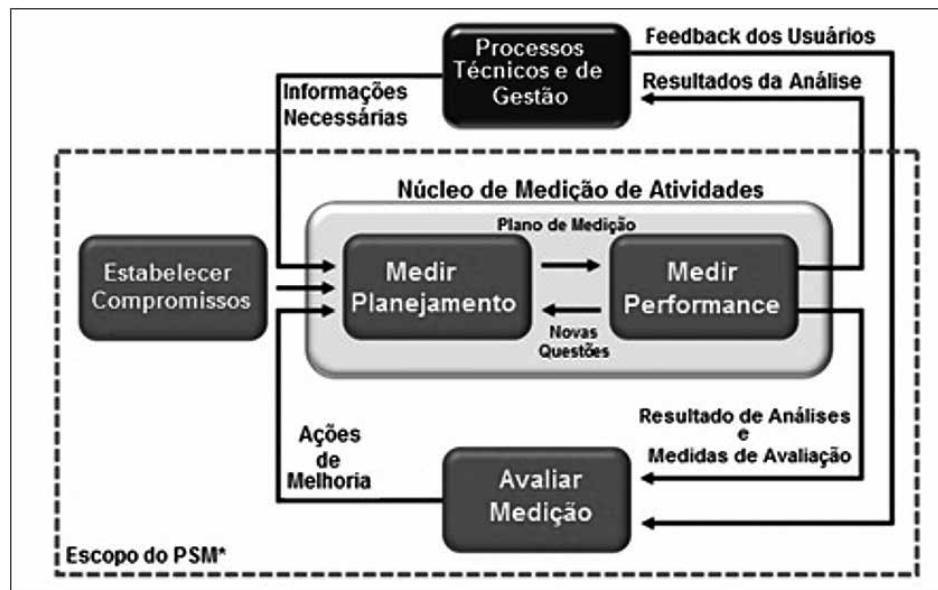
1. ISO/IEC 27000: *Information Security Management Systems – Overview and Vocabulary* (Vocabulário de Gestão da Segurança da Informação)
2. ISO/IEC 27001: *Information Security Management Systems – Requirements* (Sistema de Gestão de Segurança da Informação - Requisitos), 2013.
3. ISO/IEC 27002: *Code of Practice for Information Security Management* (Código de Práticas para Controle da Segurança da Informação), 2013.
4. ISO/IEC 27003: *Information Security Management System Implementation Guidance* (Diretrizes para Implantação de um Sistema de Gestão da Segurança da Informação), 2011.
5. ISO/IEC 27004: *Information Security Management Measurements* (Gestão da Segurança da Informação — Medição), 2010.
6. ISO/IEC 27005: *Information Security Risk Management* (Gestão de Riscos de Segurança da Informação), 2011.
7. ISO/IEC 27006: *Requirements for Bodies Providing Audit and Certification of Information Security Management Systems* (Requisitos para Organismos que Prestam Auditoria e Certificação de Sistemas de Gestão de Segurança da Informação).

FONTE: Segurança da Informação: Padrões de Segurança da Informação. Universidade Estácio de Sá. 2015. Disponível em: <<https://ajeitarakoisa.files.wordpress.com/2015/07/si-padroes-de-seguranca.pdf>>. Acesso em: 24 set. 2015.

5.6 NORMA ISO/IEC 15939

A ISO 15939 define um processo de medição aplicáveis a sistemas, engenharia de *software* e disciplinas de gestão. Conforme PSMC (2015) a norma cobre as atividades de medição, as informações necessárias, a aplicação de resultados de análises de medição, e determina se os resultados da análise são válidos. O padrão pode ser usado tanto por fornecedores de sistemas e adquirentes.

FIGURA 62 – MODELO DE PROCESSO DE MEDIÇÃO



FONTE: Disponível em: <http://sebokwiki.org/w/images/thumb/2/2b/Measurement_Process_Model-Figure_1.png/600px-Measurement_Process_Model-Figure_1.png>. Acessado em: 24 set. 2015.

O processo está descrito por meio de um modelo que define as atividades do processo de medição que são necessários para especificar adequadamente os compromissos, como as medidas e os resultados da análise devem ser aplicados, como determinar se os resultados da análise são válidos.

6 MODELOS CMMI E MPS.BR

Além dos Padrões e Normas listadas anteriormente, a área de qualidade possui diversos modelos de qualidade nas empresas de tecnologia, a seguir serão apresentados sobre o CMMI e MPS.BR, os modelos mais difundidos nas indústrias de *software* no Brasil.

6.1 CMMI (CAPABILITY MATURITY MODEL INTEGRATION): INTEGRAÇÃO DOS MODELOS DE CAPACITAÇÃO E MATURIDADE DE SISTEMAS

Há forte demanda por produtos de *software* em todos os setores da economia, mas a oferta é também ampla, e a concorrência pela melhor qualidade é acirrada. As empresas estão hoje solicitadas/exigidas a mostrar sua competência técnica, operacional e gerencial, o que requer a demonstração de controle sobre seus processos operacionais, tanto internos quanto os pertinentes às relações externas, especialmente com clientes e fornecedores.

Experiências mostram que empresas trabalham melhor quando focam seus esforços de melhoria em um número gerenciável de áreas de processos que requerem esforços cada vez mais sofisticados à medida que a organização evolui.

A respeitabilidade e amplitude de aceitação do modelo CMMI são correntemente uma exigência e um passaporte para comercialização de produtos e soluções de *software*.

Os níveis de maturidade fornecem uma maneira de prever o desempenho da organização dentro de cada disciplina ou conjunto de disciplinas. São estágios evolutivos bem definidos em busca de um processo maduro. Cada nível estabelece uma parte importante do processo da empresa. Nos modelos CMMI com representação em estágios que caracterizam o nível de capacidade do processo, existem cinco níveis de maturidade designados pelos números de 1 a 5.

Conforme tradução livre da documentação da SEI – *Software Engineering Institute* da Universidade Carnagie Mellon, segue a explicação de cada um destes níveis de maturidade:

FIGURA 63 – VISÃO GERAL DO CMMI



FONTE: Disponível em: <<http://www.dei.unicap.br/~almir/seminarios/2005.1/ns06/pmi/CMM.htm>>. Acesso em: 21 set. 2015.

CMMI é um dos modelos mais aceitos para a melhoria da qualidade e do processo de *software* em todo o mundo e define os princípios e práticas que devem ser aplicados a uma organização para atingir estágios evolutivos de maturidade em seu processo de *software*.

TABELA 16 – NÍVEIS DE MATURIDADE DO CMMI

Nível de maturidade	Características dos Processos	Descrição de cada Nível de Maturidade
1 – Inicial	Processo caótico	O processo de <i>software</i> é caracterizado com “ <i>ad hoc</i> ” e ocasionalmente pode ser caótico, não apresenta um ambiente estável possuindo a tendência de: não cumprimento da agenda estabelecida; abandono dos processos em tempo de crise; e de não serem capazes de repetir sucessos passados. Poucos processos estão definidos e o sucesso depende de esforços individuais. Seu processo é como uma “caixa preta” – não há documentação e não há controle. É um estágio onde as atividades estão mal definidas, ferramentas são usadas ao acaso ou por iniciativa pessoal. Tem-se dificuldade de visualizar e gerenciar projetos. Por fim, o cliente somente verifica se os seus requisitos foram atendidos na entrega do produto.
2 – Gerencial	Disciplinados	Os processos básicos de gerenciamento estão estabelecidos para planejar e controlar custo, cronograma e funcionalidade. O foco neste nível é mais voltado nos projetos do que na organização. Existem políticas organizacionais que orientam os projetos estabelecendo processos de gerenciamento. Compromissos são firmados, gerenciados e sucessos podem ser repetidos oferecendo assim controle e visibilidade em ocasiões definidas. Ferramentas são usadas, passa a existir gerência de configurações e de requisitos. Os projetos são bem documentados e identificados, acompanhando seus pontos de transição (“ <i>milestones</i> ”). Neste nível o cliente pode analisar o produto durante o processo de <i>software</i> (<i>checkpoint</i>). O maior desafio que as organizações enfrentam para alcançar este nível de maturidade está relacionado com a mudança cultural e não com a implantação dos novos processos propriamente ditos.
3 – Definido	Padronizados e Consistentes	Tanto para as atividades de gerência básica como para as de engenharia de <i>software</i> , o processo de <i>software</i> é documentado, padronizado e integrado num processo único, chamado Processo de <i>Software</i> Definido do Projeto. Todos os projetos usam uma versão deste processo, adaptada às características específicas do projeto, contemplando o desenvolvimento e a manutenção do <i>software</i> . As atividades são visíveis, gerentes e engenheiros entendem suas atividades e responsabilidades no processo e estão preparados proativamente para possíveis riscos. Os processos começam a serem avaliados através de inspeções e auditorias rotineiras. Os testes são os testes

		os testes são padronizados, são realizadas medições iniciais dos projetos e a gerência de configuração generalizada. O cliente pode obter status atualizado, rapidamente e corretamente, com detalhe entre as atividades.
4 – Gerenciado Quantitativa-mente	Previsíveis	Medições detalhadas do processo de <i>software</i> e da qualidade do produto são coletadas através de relatórios estatísticos. Tanto o processo de <i>software</i> quanto o produto de <i>software</i> são quantitativamente entendidos e controlados. Está estabelecido e em uso rotineiro um programa de medições, um grupo de garantia de qualidade sendo constantemente aprimorada (quantificada/planejada/avaliada). Nesta etapa se fornece aos gerentes condições de avaliar seu processo e identificar possíveis problemas, onde gerentes possuem uma base de dados para a tomada de decisões. A habilidade de prever resultados é maior e a variabilidade do processo é menor. O cliente pode estabelecer um entendimento quantitativo da capacidade do processo e riscos antes de projeto iniciar.
5 – de Otimização	Melhoria contínua	A melhoria contínua do processo é feita através do “feedback” quantitativo dos processos e das aplicações de novas ideias e tecnologias. Nesta etapa a visibilidade do processo com a melhoria contínua do processo objetividade produtividade e alto nível de qualidade, os gerentes são aptos a estimar e monitorar a eficácia da mudança e a empresa obtém forte relação de parceria com cliente.

FONTE: O autor



Hoje, mais de 3 mil organizações em todo o mundo, e mais de 100 no Brasil têm o modelo CMMI implantado e estão no nível 2 ou superior. Para maiores informações sobre seu crescimento no mundo segue aqui o *link* com os resultados da última conferência realizada pela SEI: <<http://www.sei.cmu.edu/sema/profile.html>>.

TABELA 17 – ÁREAS-CHAVE DE PROCESSOS DE ENGENHARIA DE SOFTWARE QUE UTILIZAM O MODELO CMMI

Nível	Áreas-chave de Processos
Otimização	Prevenção de Defeitos Gerenciamento de Mudanças Tecnológicas Gerenciamento de Mudanças em Processos
Gerenciado	Gerenciamento Quantitativo do Processo Gerenciamento da Qualidade do <i>Software</i>
Definido	Foco no Processo Organizacional Definição do Processo Organizacional Gerenciamento Integrado de <i>Software</i> Engenharia de Produto de <i>Software</i> Coordenação entre Grupos Revisões entre Pares (auditorias) Programa de Treinamento
Repetível	Gerenciamento de Requisitos Planejamento do Projeto de <i>Software</i> Acompanhamento e Rastreamento do Projeto Gerenciamento de Configuração de <i>Software</i> Gerenciamento de Subcontratação Garantia de Qualidade em <i>Software</i>

FONTE: O autor

A criação do CMMI por parte do SEI teve a intenção de suportar a melhoria de processos e produtos reduzindo a redundância e eliminando as inconsistências quando da utilização de modelos isoladamente. O objetivo é melhorar a eficiência, o retorno em investimento e a efetividade através da utilização de um modelo que integra disciplinas como engenharia de sistemas e engenharia de *software* que são inseparáveis no desenvolvimento de um sistema.

O principal propósito do CMMI é fornecer diretrizes baseadas em melhores práticas para a melhoria dos processos e habilidades organizacionais, cobrindo o ciclo de vida de produtos e serviços completos, nas fases de concepção, desenvolvimento, aquisição, entrega e manutenção. Neste sentido, suas abordagens envolvem a avaliação da maturidade da organização ou a capacitação das suas áreas de processo, o estabelecimento de prioridades e a implementação de ações de melhoria. (FERNANDES, 2012, p. 204).

6.2 MELHORIA DE PROCESSO DE SOFTWARE BRASILEIRO (MPS.BR)

Segundo Softex (2014), o MPS.BR é um programa que foi criado em 2003 pela própria Softex para melhorar a capacidade de desenvolvimento de *software* nas empresas brasileiras. A iniciativa foi responsável pelo desenvolvimento do Modelo de Referência para Melhoria do Processo de *Software* Brasileiro (MPS-SW), que levou em consideração normas e modelos internacionalmente reconhecidos,

boas práticas de engenharia de *software* e as necessidades de negócio da indústria de *software* nacional.

Segundo Sodré (2015), o MPS.BR possui as seguintes metas:

- definir e implementar o Modelo de Referência para Melhoria de Processos de *Software* (MR mps) em 120 empresas, até junho de 2006, com perspectiva de mais de 160 empresas nos dois anos subsequentes;
- criar cursos em diversos locais do país para capacitar e formar consultores do modelo;
- credenciar instituições e centros tecnológicos capacitados a implementar e avaliar o modelo com foco em grupo de empresas.

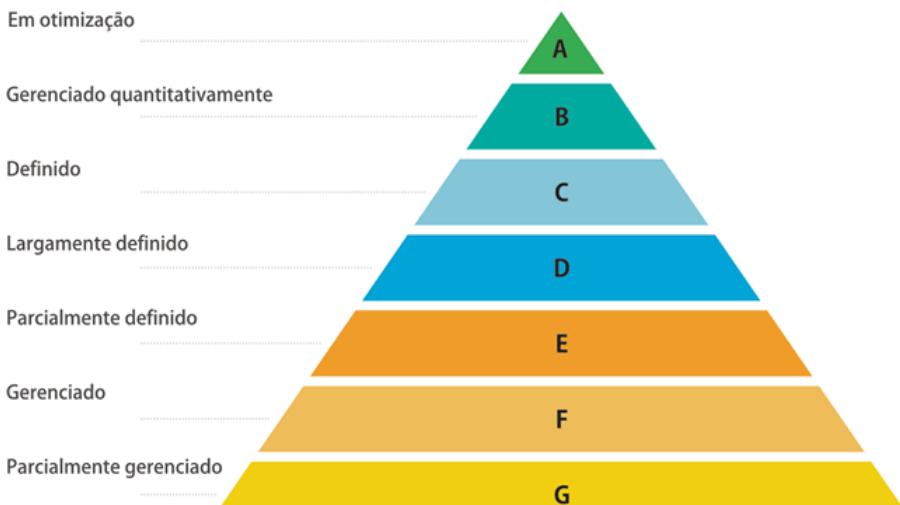
Foram criados dois modelos: o Modelo de Referência para Melhoria do Processo de *Software* (MR mps) e o Modelo de Negócio para Melhoria de Processo de *Software* (MN mps). Este modelo de negócios prevê duas situações (SODRÉ, 2015):

- a implementação do MR mps de forma personalizada para uma empresa (MNE – Modelo de Negócio Específico);
- a implementação do MR mps de forma cooperada em grupo de empresas (MNC – Modelo de Negócio Cooperado), é um modelo mais acessível às micro, pequenas e médias empresas por dividir de forma proporcional parte dos custos entre as empresas e por se buscar outras fontes de financiamento.

Na figura a seguir é possível observar os sete níveis de classificação do MPS.BR (FUMSOFT, 2015).

FIGURA 64 – NÍVEIS MPS.BR

Níveis de Qualificação



FONTE: Fumsoft. Disponível em: <http://www.fumsoft.org.br/qualidade/modelo_mpsbr>. Acesso em: 26 set. 2015.

Segundo Antonioni (2007), o MPS.BR é dividido em sete níveis de maturidade, processos e atributos, conforme exibido na tabela a seguir:

TABELA 18 – 7 NÍVEIS DE MATURIDADE, PROCESSOS E ATRIBUTOS

Níveis	Processos
A – Em otimização (mais alto)	Implantação de Inovações na Organização Análise de Causas e Resolução
B – Gerenciado Quantitativamente	Desempenho do Processo Organizacional Gerência Quantitativa do Projeto
C – Definido	Gerência de Riscos Análise de Decisão e Resolução
D – Largamente Definido	Desenvolvimento de Requisitos Solução Técnica Validação Verificação Integração do Produto
E – Parcialmente Definido	Treinamento Definição do Processo Organizacional Avaliação e Melhoria do Processo Organizacional Adaptação do Processo para Gerência de Projeto
F – Gerenciado	Gerência de Configuração Garantia da Qualidade Medição Aquisição
G – Parcialmente Gerenciado (mais baixo)	Gerência de Projeto Gerência de Requisitos

FONTE: Antonioni (2007)

O MPSBR (Melhoria do Processo de *Software* Brasileiro), que segundo Groffe (2013), foi criado no Brasil por instituições ligadas ao desenvolvimento de *software* brasileiras, entre as quais estão: Softtex(SP), RioSoft(RJ), COPPE/UFRJ (RJ) e CESAR (PE). O MPSBR, conforme o autor, tem como base práticas usadas internacionalmente para a melhoria dos processos de *software*, como o CMMI, e por isso tem compatibilidade com vários processos de qualidade, como o citado anteriormente. Assim como o CMMI o MPSBR classifica a maturidade dos processos em níveis, mas não em cinco e sim em sete níveis, do G, nível mais baixo, ao A, nível mais alto, e a proposta para cada nível é, de acordo com Groffe (2013):

- **G – Parcialmente Gerenciado:** neste ponto deve-se iniciar o gerenciamento de requisitos e projetos;
- **F – Gerenciado:** introduz controles de mediação, gerência de configuração, conceitos de aquisição e garantia de qualidade;
- **E – Parcialmente Definido:** considera processos como treinamento, adaptação de processos para gerência de projetos, além da preocupação com a melhoria e o controle do processo organizacional;

- **D – Largamente Definido:** envolve validação, verificação, liberação, instalação e integração de produtos e outras atividades;
- **C – Definido:** ocorre a gerência de riscos;
- **B – Gerenciado Quantitativamente:** avalia-se o desempenho dos processos e a gerência quantitativa dos mesmos;
- **A – Otimização:** preocupação com a inovação e análise de causas.

6.3 COMPARAÇÃO CMMI E MPS.BR

Ambos os modelos possuem níveis de maturidade que definem a capacidade das empresas em trabalhar com projetos grandes e complexos. O CMMI por sua vez possui os níveis do 1 ao 5 e o MPS.BR possui os níveis do G ao A, sendo que ao contrário do CMMI, o primeiro nível do MPS.BR já exige que a empresa tenha determinados processos identificados (ISTITUTO DE SOFTWARE, 2015).

A tabela a seguir representa a correlação entre CMMI e MPS.BR, respectivamente, nela estão caracterizadas as principais semelhanças contidas nos dois modelos.

TABELA 19 – CORRELAÇÃO ENTRE OS MODELOS CMMI E MPS.BR

CMMI		MPS.BR	
5	Análise Casual e Resolução Inovação e Melhoria Organizacional	A	Análise de Causas de Problemas e Resolução
4	Desempenho do Proc. Org. Quantitativa de Projeto	B	Gerência Quantitativa do Projeto
3	Foco no Processo da Organização Definição do Proc. da Organização Treinamento Organizacional Gerência Integrada de Projeto Gerência de Risco Desenvolvimento de Requisitos Solução Técnica Integração de Produto Verificação Validação Análise de Decisão e Resolução	C	Análise de Decisão e Resolução Gerência de Riscos Desenvolvimento de Reutilização
		D	Desenvolvimento de Riscos Integração do Produto Projeto e Construção do Produto Verificação Validação
		E	Gerência de Recursos Humanos Avaliação e Melhoria do Proc. Org. Definição do Proc. Organizacional Gerência de Reutilização
2	Gerência de Requisitos Planejamento de Projeto Acompanhamento e Contr. de Proj. Ger. de Acordo com Fornecedores Gar. de Qual. de Proc. e Produto Gerência de Configuração Medição e Análise	F	Medição Gerência de Configuração Aquisição Garantia da Qualidade
		G	Gerência de Requisitos Gerência de Projetos

FONTE: Franciscani (2015)

Segundo Franciscani (2015 apud OLIVEIRA, 2008) existem medições entre os modelos e as comparações entre eles podem ser visualizadas na tabela a seguir.

TABELA 20 – COMPARATIVO ENTRE OS MODELOS CMMI E MPS.BR

CMMI	MPS.BR
O modelo de Qualidade CMMI é reconhecido internacionalmente.	O MPS.BR é mais conhecido nacionalmente e na América Latina.
O modelo CMMI envolve um grande custo na avaliação e certificação do Modelo.	No MPS.BR o custo da certificação é mais acessível.
No CMMI é necessário investir tempo, geralmente para se chegar aos níveis de maturidade mais altos.	No MPS.BR as avaliações são bienais.
O CMMI tem foco global voltado para empresas de maior porte.	MPS.BR é um modelo criado em função das médias e pequenas empresas.
O CMMI possui cinco níveis de maturidade por estágio e seis na contínua.	MPS.BR possui sete níveis de maturidade, onde a implementação é mais gradual.
O CMMI é aceito como maturidade para licitações.	O MPS.BR é o aceito como maturidade para licitações.
O CMMI torna as empresas competitivas internacionalmente.	O MPS.BR não torna as empresas competitivas internacionalmente.
O CMMI não utiliza contrato conjunto de empresas.	No MPS.BR pode acontecer contrato cooperado em grupo de empresas que queiram a certificação.
Implementação mais complexa.	Implementação mais simples.
Desenvolvido pelo Software Engineering Institute – SEI em 1992.	Desenvolvido por algumas instituições Brasileiras em 2003.

FONTE: Franciscani (2015)

RESUMO DO TÓPICO 1

Neste tópico, você aprendeu que:

- Em desenvolvimento de *software*, a qualidade significa atendimento aos requisitos do cliente, projeto com zero defeito, aumento de produtividade, redução de custos e uma boa usabilidade do Sistema.
- Para Roger Pressman (2011), qualidade é conformidade a requisitos funcionais e de desempenho explicitamente declarados, a padrões de desenvolvimento claramente documentados e a características implícitas que são esperadas de todo *software* profissionalmente desenvolvido.
- Joseph M. Juran fundamentou a abordagem de qualidade em três processos básicos:
 - ✓ Planejamento da Qualidade: definição da qualidade;
 - ✓ Controle da Qualidade: garantir o grau de qualidade e produtividade previamente estabelecido;
 - ✓ Melhoria da Qualidade: a identificação de oportunidades de melhoria e divulgação de resultados.
- Uma das principais formas de implementação do controle qualidade é a utilização do Ciclo PDCA (*Plan-Do-Check-Action*), que consiste em quatro fases: planejar, fazer, checar e agir corretamente:
 - ✓ Na fase *Plan*, o foco está na identificação do problema, análise do processo atual e definição do plano de ação para melhoria do processo em questão.
 - ✓ Na fase *Do*, o plano de ação definido deve ser executado e controlado.
 - ✓ Em *Check*, verificações devem ser realizadas, a fim de subsidiar ajustes e se tirar lições de aprendizagem.
 - ✓ No *Action* deve-se atuar corretivamente para fundamentar um novo ciclo, garantindo a melhoria contínua.
- *Total Quality Management* (TQM) dividiu o desenvolvimento de *software* em dois momentos:
 - ✓ A fase do *software* artesanal: antes da definição de Engenharia de *Software* em 1969, o *software* era em geral desenvolvido e testado continuamente até se chegar a algo que funcionasse e que o cliente pudesse aceitar.
 - ✓ A fase do *software* profissional: já a era da Engenharia de *software*, além de funcionar e ser aceito pelo cliente, o *software* precisava ser padronizado, documentado e com boa relação custo/benefício.

- Para produzir um produto de *software* com qualidade devem-se possuir processos formais que visem à prevenção e detecção de defeitos durante o desenvolvimento de *software*.
- Todo processo de *software* deve possuir junto ao plano de projeto uma documentação específica da qualidade, denominada como plano de qualidade que deve compreender informações sobre como a equipe de qualidade irá garantir o cumprimento da política de qualidade, no contexto do programa ou projeto a serem desenvolvidos, quais métodos, técnicas, métricas, treinamentos e padrões devam ser utilizados ao longo de todo o ciclo de vida do projeto. O plano deve oferecer a base do gerenciamento dos riscos, dos testes, das inspeções, das auditorias e como deverão ocorrer os reportes de problemas e ações corretivas.
- A Garantia da Qualidade de *Software* é a definição padronizada das atividades voltadas a prevenção de defeitos e problemas, que podem surgir nos produtos de trabalho. Área que define padrões, metodologias, técnicas e ferramentas de apoio ao desenvolvimento tendo como entrada o plano de qualidade de *software* e os resultados de medições de qualidade.
- O Controle de Qualidade de *Software* é voltado para o monitoramento de resultados específicos do projeto, ou seja, a detecção de defeitos, executadas através do uso de técnicas que incluem revisões por pares, teste e análise de tendências, entre outras. Abaixo consta a tabela com as principais diferenças entre elas.
- O custo de qualidade é categorizado em custos de prevenção, custos de avaliação, custos de falhas internas e custos de falhas externas a fim despender mais esforço em prevenção e detecção para assim reduzir defeitos.
- Para se obter um produto com qualidade, uma série de fatores são necessários: a qualidade do processo para um bom planejamento do projeto, a tecnologia de desenvolvimento utilizada através de ferramentas para construir o *software*, custo, tempo e cronograma implicam nos recursos financeiros e tempo que terá para fazer o *software*, sendo estes os principais limitadores devido recursos financeiros insuficientes e prazos muitos curtos. Por último tem-se a qualidade das pessoas, que é o fator decisivo, uma equipe bem qualificada realiza um trabalho de qualidade.
- Diversos padrões e normas de qualidade de *software* vêm sendo propostos ao longo dos anos. Essas normas têm sido fortemente adotadas nos processos de *software* das organizações em todo o mundo.
- As principais normas ISO aplicadas à qualidade de produto ou processo de *software* são:
 - ✓ Norma ISO/IEC 9000: é um conjunto de documentos que engloba pontos referentes à garantia da qualidade em projeto, desenvolvimento, produção, instalação e serviços associados; objetivando a satisfação do cliente pela

prevenção de não conformidades em todos os estágios envolvidos no ciclo da qualidade da empresa

- ✓ Norma ISO/IEC 12207: é o estabelecimento de uma estrutura comum utilizada como referência para os processos de ciclo de vida de *software* considerando que o desenvolvimento e a manutenção de *software* devem ser conduzidos da mesma forma que a disciplina de engenharia.
 - ✓ Norma ISO/IEC 15504: possui um conjunto de nove documentos que endereçam avaliação de processo, assessoria de treinamento e competência, determinação da capacidade e melhoria de processo e está se tornando um modelo de referência para outros padrões como o CMMI.
 - ✓ Norma ISO/IEC 9126: estabelece um modelo de qualidade para o produto de *software* que são avaliados conforme seis categorias básicas que são subdivididas em algumas características que são importantes para cada categoria: funcionalidade, confiabilidade, eficiência, usabilidade, manutenibilidade e portabilidade.
 - ✓ Norma ISO/IEC 27000: trata sobre a área de segurança da informação através de Requisitos do SGSI; Controles de Segurança; Diretrizes de Implementação; Medição; Gestão de Risco e Auditoria de Segurança.
 - ✓ Norma ISO/IEC 15939: define um processo de medição aplicável a sistemas, engenharia de *software* e disciplinas de gestão.
-
- Os modelos de qualidade mais difundidos nas indústrias de *software* no Brasil são o CMMI e MPS.BR.
 - O principal propósito do CMMI (*Capability Maturity Model Integration* ou Integração dos Modelos de Capacitação e Maturidade de Sistemas) é fornecer diretrizes baseadas em melhores práticas para a melhoria dos processos e habilidades organizacionais, cobrindo o ciclo de vida de produtos e serviços completos, nas fases de concepção, desenvolvimento, aquisição, entrega e manutenção.
 - CMMI é um dos modelos mais aceitos para a melhoria da qualidade e do processo de software em todo o mundo e define os princípios e práticas que devem ser aplicados a uma organização para atingir estágios evolutivos de maturidade em seu processo de *software*. Os cinco níveis de maturidade são: (1) inicial: Processo imprevisível e sem controle. (2) Repetível: processo disciplinado. (3) Definido: processo consistente e padronizado. (4) Gerenciado: processo previsível e controlado e (5) Otimização: processo aperfeiçoado continuamente.
 - O MPS. BR (Melhoria de Processo de *Software* Brasileiro) é um programa que foi criado para melhorar a capacidade de desenvolvimento de *software* nas empresas brasileiras.
 - MPS.BR possui as seguintes metas: (1) definir e implementar o Modelo de Referência para Melhoria de Processos de *Software* (MR mps). (2) criar cursos

em diversos locais do país para capacitar e formar consultores do modelo. (3) credenciar instituições e centros tecnológicos capacitados a implementar e avaliar o modelo com foco em grupo de empresas.

- Os sete níveis de maturidade do MPS.Br são: (G) Parcialmente gerenciado: inicia o gerenciamento de requisitos e projetos; (F) Gerenciado: inclui medições, gerência de configurações e garantia de qualidade; (E) Parcialmente definido: inclui treinamento, adaptação de processos para gerência de projetos; (D) Largamente definido: envolve teses e integração de produto; (C) Definido: gerência de riscos; (B) Gerenciado quantitativamente: avalia o desempenho dos processos e a gerência quantitativa dos mesmos; e (A) em otimização: preocupação com a inovação e análise de causas.

AUTOATIVIDADE



1 Um dos fundamentos atuais da qualidade é:

- a) () estar em conformidade com as expectativas do cliente.
- b) () possuir competências específicas de gestão.
- c) () definir o padrão do produto a partir da perspectiva originada pelo gerente da área de qualidade da organização.
- d) () buscar a maximização dos lucros como foco prioritário.

2 Apesar de todas as afirmativas abaixo definirem corretamente o que é “Qualidade de *Software*”, qual resposta corresponde à definição dada por Roger Pressman, um dos gurus da Engenharia de *Software*?

- a) () Significa atendimento aos requisitos; qualidade vem através de prevenção; padrão para desempenho da qualidade e “defeitos zero”; a medida de qualidade é o preço da não conformidade.
- b) () Conformidade a requisitos funcionais e de desempenho explicitamente declarados, a padrões de desenvolvimento claramente documentados e a características implícitas que são esperadas de todo software profissionalmente desenvolvido.
- c) () Qualidade de *software* é um processo sistemático que focaliza todas as etapas e artefatos produzidos com o objetivo de garantir a conformidade de processos e produtos, prevenindo e eliminando defeitos.
- d) () Possui como princípio básico tentar prevenir defeitos ao invés de consertá-los, ter a certeza que os defeitos que foram encontrados, sejam corrigidos o mais rápido possível, estabelecer e eliminar as causas, bem como os sintomas dos defeitos e auditar o trabalho de acordo com padrões e procedimentos previamente estabelecidos.

3 Ciclo PDCA inclui as seguintes etapas sequenciais:

- a) () Diagnóstico; definição de metas; monitoramento; avaliação.
- b) () Planejamento; execução; controle/verificação; ação avaliativa/corretiva.
- c) () Priorização; definição de objetivos, capacitação; ação avaliativa/corretiva.
- d) () Planejamento; desenvolvimento; capacitação; avaliação.

4 Para produzir um produto de *software* com qualidade deve-se possuir processos formais que visem à prevenção e detecção de defeitos durante o desenvolvimento de *software*. A respeito dos exemplos destas duas técnicas, assinale V para verdadeiro ou F para falso:

- () A técnica de prevenção de defeitos em um processo de desenvolvimento de *software* se dá pelo uso de instruções de procedimentos (padrões formais), treinamentos, documentação, modelagem e reengenharia.

- () A técnica de prevenção de defeitos em um processo de desenvolvimento de *software* podem ser pela análise de código; revisão por pares; testes, auditorias, verificações e validações.
- () As técnicas de detecção de defeitos em um processo de desenvolvimento de *software* podem ser pela análise de código; revisão por pares; testes, auditorias, verificações e validações.
- () As técnicas de detecção de defeitos em um processo de desenvolvimento de *software* se dá pelo uso de instruções de procedimentos (padrões formais), treinamentos, documentação, modelagem e reengenharia.

A sequência correta é:

- a) () F – V – V – F.
- b) () V – F – V – F.
- c) () V – V – V – F.
- d) () V – F – F – V.

5 Na gestão da qualidade de *software* existem diversas atividades voltadas à garantia da qualidade e ao controle de qualidade de *software*. A respeito das suas definições assinale V para verdadeiro ou F para falso:

- () A garantia da qualidade é para a definição padronizada das atividades voltadas à prevenção de defeitos e problemas, que podem surgir nos produtos de trabalho.
- () A garantia da qualidade é área que define padrões, metodologias, técnicas e ferramentas de apoio ao desenvolvimento tendo como entrada o plano de qualidade de *software* e os resultados de medições de qualidade.
- () O controle de qualidade é voltado para o monitoramento de resultados específicos do projeto, ou seja, a detecção de defeitos, executadas através do uso de técnicas que incluem revisões por pares, teste e análise de tendências, entre outras.
- () O controle de qualidade é área que define padrões, metodologias, técnicas e ferramentas de apoio ao desenvolvimento tendo como entrada o plano de qualidade de *software* e os resultados de medições de qualidade.

A sequência correta é:

- a) () F – V – V – V.
- b) () V – F – F – V.
- c) () V – V – V – F.
- d) () V – V – F – F.

6 A respeito da Norma ISO/IEC 9000 qual é a resposta correta:

- a) () É um conjunto de normas, que só podem ser utilizadas por empresas grandes de caráter industrial.
- b) () É um pacote de softwares orientado para implantação de sistemas de qualidade em empresas do setor de informática.

- c) () Confere qualidade a um produto (ou serviço), garantindo que o produto (ou serviço) apresentará sempre as mesmas características.
d) () Diz respeito apenas ao sistema de gestão da qualidade de uma empresa, e não às especificações dos produtos fabricados por esta empresa.

7 A norma NBR ISO/IEC 12207 estabelece:

- a) () Os processos fundamentais, organizacionais e de apoio do ciclo de vida de *software*.
b) () As atividades de tecnologia da informação agrupadas em processos e esses em domínios.
c) () Os estágios do ciclo de vida dos serviços de tecnologia da informação.
d) () Um modelo de áreas de processos representadas por categoria e por estágios.

8 De acordo com a norma ISO/IEC 9126, a qualidade do produto *software* está relacionada às seguintes características: Funcionalidade, Confiabilidade, Usabilidade, Eficiência, Manutenibilidade e Portabilidade. Sobre o tema, assinale a afirmação correta.

- a) () A Manutenibilidade diz que o produto de *software* deve ser capaz de manter seu nível de desempenho, ao longo do tempo, nas condições estabelecidas.
b) () A Confiabilidade está relacionada ao esforço necessário para a utilização do sistema, baseado em um conjunto de implicações e de condições do usuário.
c) () A Usabilidade refere-se à compatibilidade dos recursos e os tempos envolvidos compatíveis com o nível de desempenho requerido pelo *software*.
d) () A Funcionalidade refere-se à existência de funções e propriedades específicas do produto, que satisfazem as necessidades do usuário.

9 Em relação às normas da família ISO 27000, correlacione as colunas a seguir considerando a descrição que melhor define cada norma:

Norma	Descrição
I. ISO 27001	Métricas para avaliação de um sistema de gestão de segurança da informação.
II. ISO 27002	Guia para implementação de sistemas de gestão de segurança da informação.
III. ISO 27003	Boas práticas para gestão de segurança da informação.
IV. ISO 27004	Especificação para implementação de um sistema de gestão de segurança da informação.

Está CORRETA a seguinte sequência de respostas:

- a) () IV, III, II, I.
- b) () I, III, IV, II.
- c) () III, IV, II, I.
- d) () IV, I, II, III.

10 Os Níveis de Maturidade de 1 a 5 do CMMI são:

- a) () Inicial, Projetado, Definido, Gerenciado Qualitativamente e Otimização.
- b) () Inicial, Gerenciado, Dirigido, Verificado Quantitativamente e Maximizado.
- c) () Inicial, Repetível, Definido, Gerenciado Quantitativamente e Otimização.
- d) () Inicial, Repetível, Gerenciado, Revisto, Otimizado e Quantificado.

11 O MPS.BR consiste simultaneamente em um movimento para a Melhoria de Processo do *Software* Brasileiro (programa MPS.BR) e um modelo de qualidade de processo (modelo MPS) direcionado para pequenas e médias empresas de desenvolvimento de *software* no Brasil. Sobre o MPS.BR, sabe-se também que ele:

- a) () Carece de um método de avaliação para melhoria de processo de *software*.
- b) () Possui incompatibilidade com o modelo de referência CMMI (*Capability Maturity Model Integration*).
- c) () Tem alto custo de certificação em relação às normas estrangeiras.
- d) () Apresenta sete níveis de maturidade (do nível A ao G), cada qual com suas áreas de processo, onde são analisados processos fundamentais, organizacionais e de apoio.

MÉTODOS ÁGEIS

1 INTRODUÇÃO

Na segunda metade da década de 90 houve uma reação da indústria do *software* contra as dificuldades encontradas nos métodos clássicos, como a demora no desenvolvimento, a inflexibilidade e falta de qualidade no *software*. Com as demandas e concorrências subindo, os engenheiros de *software* necessitaram mais uma vez inovar, criando uma metodologia que usasse uma forma ágil de desenvolver, diminuindo os custos e minimizando erros no *software*.

Surgiram assim as metodologias leves, mudando em 2001 para metodologias ágeis, quando proeminentes do *software* se reuniram e desenvolveram o manifesto ágil, que traz as principais regras, princípios e práticas das metodologias ágeis. O manifesto contém quatro principais valores, sendo eles (MANIFESTOAGIL, 2015):

- A capacidade de respostas às mudanças e flexibilidade do *software* acima de um plano pré-estabelecido;
- A colaboração e participação dos clientes acima das negociações e contratos;
- Os indivíduos e suas interações acima das ferramentas e procedimentos;
- O cumprimento dos requisitos e funcionamento do *software* acima de documentação complexa.

Os modelos ágeis surgem como uma reação natural à expansão do CMMI no mercado mundial, atingindo não apenas as grandes organizações, mas também pequenas e médias empresas de TI (BARTIE, 2015).

As Metodologias Ágeis de Desenvolvimento de *Software* são indicadas como sendo uma opção às abordagens tradicionais para desenvolver *softwares*: produzem pouca documentação, é recomendado documentar somente o que será útil. Em essência, as Metodologias Ágeis foram desenvolvidas com o objetivo de vencer as fraquezas percebidas e reais da Engenharia de *Software* (PRESSMAN, 2010).

São recomendadas para projetos que: existem muitas mudanças; os requisitos são passíveis de alterações; a recodificação do programa não acarreta alto custo; a equipe é pequena; as datas de entrega curtas acarretam alto custo; o desenvolvimento rápido é fundamental (SOUZA, 2015).

Segundo Ambler (2003), o desenvolvimento ágil de *software* tem como princípios priorizar o cliente mediante entregas de *software* de valor em tempo hábil permitindo continuamente receber mudanças de requisitos, onde as equipes de negócios e de desenvolvimento trabalham juntas e motivadas, diariamente durante todo o projeto buscando que a equipe reflita sobre como se tornar mais eficaz ajustando e adaptando seu comportamento.

A modelagem ágil é um conjunto de práticas guiado por princípios e valores para profissionais de *software* aplicarem no dia a dia e tem como objetivos definir e mostrar como colocar em prática um conjunto de valores, princípios e práticas relativas a uma modelagem eficaz e leve. Lida com a questão de como aplicar técnicas de modelagem em projetos de *software* adotando uma perspectiva ágil. Discute como é possível melhorar as atividades de modelagem.

As metodologias ágeis privilegiam o conhecimento tácito das pessoas, interações entre os indivíduos ao invés de processos e ferramentas. Este fator pode beneficiar diretamente a organização baixando o custo de seus processos de engenharia de *software*.

O desenvolvimento de *software* começou com linguagem de máquina, após isso passou para linguagens simbólicas como o Assembly, passando para linguagens estruturadas como o Fortran, chegando até a orientação a objetos. Junto com as linguagens as metodologias de desenvolvimento de *software* também evoluíram. O que possibilitou o surgimento das metodologias de desenvolvimento de *software* foi a crise de *software* de 1970, onde a engenharia de *software* era praticamente inexistente.

As principais causas da crise de *software* nos últimos quarenta anos foram o não cumprimento dos prazos dos projetos, o não cumprimento dos requisitos pré-estabelecidos, a baixa qualidade nos softwares, projetos estourando orçamentos e os códigos não gerenciáveis e ilegíveis.

Em meio a todas essas dificuldades, a engenharia de *software* até então imatura, deparou-se com o problema de reduzir os problemas enfrentados na crise, e a saída encontrada foi a padronização no desenvolvimento de *software*.

As primeiras metodologias desenvolvidas são conhecidas como metodologias clássicas, as quais basicamente são caracterizadas pela rigidez e altos níveis de controle, na qual os requisitos são altamente conhecidos, o desenvolvimento é dividido em etapas que geram muita documentação, devendo ser aprovada para seguir adiante. Embora ainda sejam muito utilizadas, as metodologias clássicas surgiram em uma época onde a manutenção no *software* era de custo altíssimo, tornando necessário todo um planejamento e documentação antes do desenvolvimento, evitando assim a maior parte dos erros no *software*. A metodologia clássica mais utilizada até hoje é a cascata, caracterizada pelas etapas distribuídas de forma linear.

2 CARACTERÍSTICAS DOS MÉTODOS ÁGEIS

Embora muito eficiente, nem sempre é fácil sua aplicação, sendo que há características em comum nas empresas que adotaram os métodos ágeis, tornando-se diretrizes para sua utilização. Pode-se exemplificar um ambiente com condições adequadas para a utilização dos métodos ágeis, um ambiente onde a criticidade é baixa, as mudanças dos requisitos são frequentes, onde há um pequeno número de desenvolvedores, porém todos muito eficientes, cultura que tem sucesso no caos.

Em contrapartida, ambientes com alta criticidade, contendo um grande número de desenvolvedores, sendo vários deles inexperientes, com poucas alterações nos requisitos, são grandes candidatos à utilização das metodologias clássicas.

A tabela a seguir apresenta um resumo comparativo dos métodos ágeis com o modelo tradicional.

TABELA 21 – PRINCIPAIS DIFERENÇAS ENTRE MODELOS ÁGEIS E CONTROLADOS

Características	Modelo Ágil	Modelo Controlado
Premissa Fundamental	Ênfase na Agilidade	Ênfase no Controle Operacional
Condução dos Trabalhos	Baseado em Processos Empíricos	Baseado em Processos Formais
Escopo da Solução	Centradas no Desenvolvimento	Englobam todas as Disciplinas
Profundidade da Abordagem	Definir apenas o que deve ser feito	Definir o que e como deve ser feito
Foco dos Profissionais	Atuação Local (por projeto)	Atuação Global (por disciplina)
Abordagem Estratégica	Atender melhor o Curto Prazo	Atender melhor o Longo Prazo
Palavras-Chaves	Pessoas, FeedBack, Adaptação	Maturidade, Estrutura, Padronização
Modelos de Implementação	XP, SCRUM, FDD, APM, Lean, Crystal e DSDM	CMMI, RUP, ITIL, ISO, PMI, MPS.br
Frases que resumem sua Filosofia	Aproxime sua equipe do Cliente, simplifique o projeto e aumente sua produtividade	Não podemos melhorar o que não podemos controlar

FONTE: Bartie (2015)

Conforme descrito na tabela comparativa, identificam-se diversos pontos fortes e fracos que contribuem para o processo de decisão estratégica da empresa de qual modelo utilizar.

3 PRINCIPAIS MÉTODOS ÁGEIS

Conforme Souza (2015), em 2001, Kent Beck e mais 16 desenvolvedores, produtores e consultores de *software*, que formavam a Aliança Ágil, assinaram o “Manifesto de Desenvolvimento Ágil” de *Software*, declarando: **Estamos descobrindo melhores maneiras de se desenvolver software, fazendo isto e ajudando os outros a fazer isto. Através deste trabalho, nós passamos a valorizar:**

- Indivíduos e interações mais que processos e ferramentas;
- Software funcionando mais que documentação abrangente;
- Colaboração com o cliente mais que negociação de contratos;
- Responder à mudança mais que seguir um plano.

Portanto, apesar de existir valor nos itens à direita, valorizamos mais os itens à esquerda.

Segundo Souza (2015), os 12 princípios do Manifesto Ágil são:

1. Garantia da satisfação do consumidor com entrega rápida e contínua de *softwares* funcionais.
2. Mudanças de requisitos, mesmo no fim do desenvolvimento, ainda são bem-vindas.
3. Frequentemente são entregues *softwares* funcionais (semanas, ao invés de meses).
4. Desenvolvedores e pessoas relacionadas aos negócios devem trabalhar, em conjunto, até o fim do projeto.
5. Construir projetos com indivíduos motivados, dar-lhes ambiente e suporte necessários e confiar que farão seu trabalho.
6. Uma conversa face a face é o método mais eficiente e efetivo de transmitir informações para e dentro de uma equipe de desenvolvimento.
7. *Software* em funcionamento é a principal medida de progresso.
8. Desenvolvimento sustentável, de modo a manter um ritmo constante indefinidamente.
9. Atenção contínua para com a excelência técnica e para com bons projetos aumenta a agilidade.
10. Simplicidade – a arte de maximizar a quantidade de trabalho não efetuado – é essencial.
11. As melhores arquiteturas, requisitos e projetos emergem de equipes auto-organizáveis.
12. Em intervalos regulares, a equipe deve refletir sobre como se tornar mais eficiente.

Entre todos os métodos ágeis podem-se citar como exemplo o *Scrum*, *Extreme Programming*, *Adaptive Software Development* (ASD), *Dynamic System Development Method* (DSDM), *Crystal Clear*, *Feature-Driven Development* (FDD) entre outros. As variações entre estes métodos se dão pela ênfase e modo de aplicação dos princípios dos métodos ágeis, que serão apresentadas a seguir.

3.1 SCRUM

Scrum é um método ágil de desenvolvimento de *software* criado por Jeff Sutherland e sua equipe no início de 1990. Recentemente, foram realizados desenvolvimentos adicionais nos métodos gráficos *Scrum* por Schwaber e Beedle (PRESSMAN, 2011).

O *Scrum* considera uma abordagem mais humana ao solucionar os problemas existentes no desenvolvimento de *Software*. Ao invés de desperdiçar tempo criando documentações extensas e detalhadas que as pessoas acabam não lendo minuciosamente. No *Scrum*, as equipes trabalham com *sprints*. São realizadas reuniões curtas onde o time verifica quais as decisões que devem ser tomadas e os recursos do *product backlog* que entram nos *sprints*. Elas também decidem quem trabalha nos *sprints* e quanto tempo dura cada tarefa (DIMES, 2014).

Um líder de equipe chamado *Scrum Master*, conduz a reunião e avalia as respostas de cada integrante. A reunião *Scrum*, realizada diariamente, ajuda a equipe a revelar problemas potenciais o mais cedo possível (PRESSMAN, 2011).

Outro papel fundamental na metodologia é o *Product Owner*, ele é o dono do produto. Fornece o requisito do negócio para a equipe assim como sua ordem de aplicação. Ou seja, o *Product Owner* é a interface entre a empresa e os clientes. É ele o ponto de contato para esclarecimento das dúvidas da equipe sobre as regras do produto. Trabalhando em conjunto com a equipe, ele ajuda a definir a ordem de execução das atividades conforme a necessidade do cliente, definindo também o cronograma para a liberação e fazendo as validações necessárias (RODRIGUES, 2015).

A equipe ou time, no *framework Scrum*, é multidisciplinar e é composta por pessoas que fazem o trabalho de desenvolvimento e teste do produto. A equipe é responsável pelo desenvolvimento do produto, e também tem a autonomia para tomar decisões de como executar o seu trabalho. Os membros da equipe decidem como dividir o trabalho em tarefas, e ao longo da *Sprint* decidem a ordem de execução das tarefas. Nove pessoas na equipe é a quantidade ideal para uma boa comunicação e não afetar a produtividade (RODRIGUES, 2015).

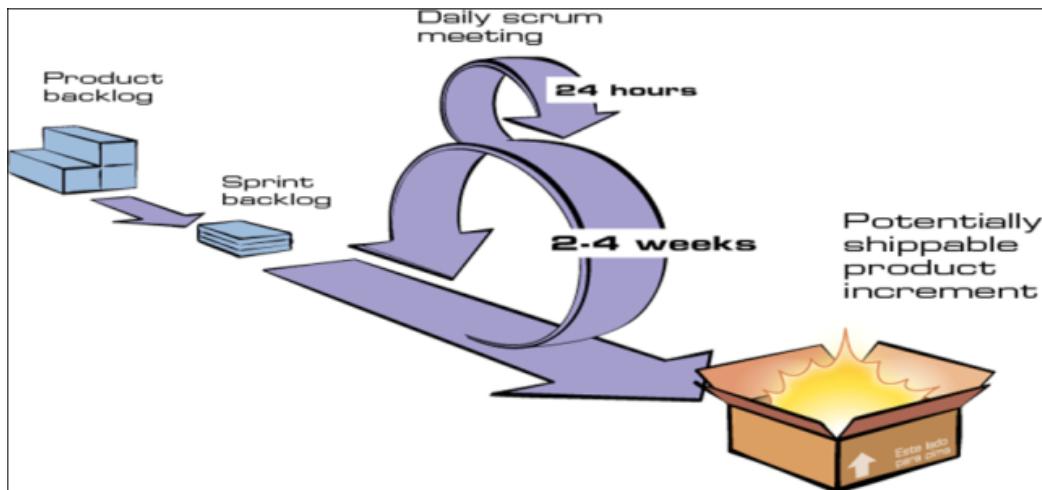
Possivelmente é o método mais utilizado hoje em dia. O *Scrum* serve como fundamento para um projeto complexo, não ditando regras que devem ser estritamente seguidas, mas que podem ser personalizadas conforme a necessidade da equipe, servindo como base para uma gerência de sucesso.

O *Scrum* é um método orientado a iterações, sendo elas chamadas de *Sprints*. As entradas de *sprints* ocorrem normalmente uma vez por mês. Os requisitos e funcionalidades a serem desenvolvidas em um determinado projeto são armazenados em uma lista conhecida como *Product Backlog*. Ao iniciar-se um *Sprint*, é realizada uma reunião de planejamento, na qual o *Product Owner* dita quais as principais funcionalidades a serem implementadas e à equipe as atividades que será capaz de produzir. As funcionalidades que serão implementadas em um *Sprint* são removidas do *Product Backlog* e alocadas no *Sprint Backlog*.

A cada dia é realizada uma reunião, analisando o que foi produzido no dia anterior, e o que será produzido no dia atual. Essa reunião é chamada de *Daily Scrum* e acontece normalmente no início da manhã.

Ao fim de um *Sprint*, a equipe apresenta os requisitos e funcionalidades desenvolvidas em uma *Sprint Review Meeting*. Após uma retrospectiva a equipe de desenvolvimento passa para o planejamento do próximo *Sprint*. E assim segue o ciclo, conforme a figura a seguir.

FIGURA 65 – PROCESSO SCRUM



FONTE: Processo Scrum. Disponível em: <http://videos.web-03.net/artigos/Cazuza_Neto/Conhecendo_Scrum/Conhecendo_Scrum1.jpg>. Acesso em: 26 set. 2015.

A divisão de tarefas no *Scrum* baseia-se em *Sprints* e Reuniões Diárias. O *Sprint* é o ciclo em que diversas atividades precisam ser feitas e entregues no final para que possam ser entregues para o cliente, possuem duração fixa, normalmente de duas a quatro semanas, mas podem ser adaptadas de acordo com a necessidade da empresa, desde que mantenha a entrega constante.

Para ajudar a manter o time atualizado, é comum no *Scrum* que todos os dias, no mesmo horário seja realizada uma breve reunião em pé, que consiste em

três pequenas perguntas: “O que fiz ontem em relação ao projeto?”, “O que vou fazer hoje em relação ao projeto?”, “Existe algum impedimento para que a meta do *Sprint* seja alcançada?”, ao responder estas perguntas, o time saberá como está indo o andamento do projeto.



Para tornar-se atualizado quanto ao modelo *Scrum*, esse é o ponto de encontro feito pela comunidade, para a comunidade de *Scrum* desse Brasil e do mundo. Disponível em: <<http://www.scrum.org.br/>>. Boa leitura!

3.2 EXTREME PROGRAMMING

De acordo com Myers (2004), por volta dos anos 90, Kent Beck iniciou uma nova abordagem ao desenvolvimento de *software* num projeto chamado C3. Essa nova metodologia visava deixar o projeto mais leve, garantir que testes fossem feitos e refeitos, melhorar a comunicação entre os membros da equipe e entre os desenvolvedores e o cliente. Desse projeto, surgiu o *Extreme Programming* (Programação Extrema).

Em relação a isso, Sommerville (2011) afirma que nesse método a diferença está na forma como o sistema é testado. Não existe especificação do sistema que possa ser usada por uma equipe de teste externa. Para evitar problemas nos testes a abordagem XP enfatiza a importância dos testes do programa, incluindo um foco de testes que reduz as chances de erros não identificados na versão atual do sistema. Existem características próprias de testes que são:

- **Desenvolvimento em *test-first*:** são escritos primeiramente os testes, depois os códigos;
- **Desenvolvimento de teste incremental a partir de cenários:** os cenários ou estórias são os requisitos de sistema, e quem os prioriza para serem desenvolvidos é o usuário;
- **Envolvimento dos usuários no desenvolvimento de testes e validação:** nesse modelo de desenvolvimento o papel do cliente passa a ser fundamental, pois ele ajuda a desenvolver os testes de aceitação, na prática, em vez de um único teste, é realizada uma bateria de testes de aceitação. Como conseguir o apoio do cliente normalmente é mais complicado, essa é uma grande dificuldade do processo de teste XP;
- **Uso de frameworks de testes automatizados:** os testes automatizados são escritos como componentes executáveis antes que a tarefa seja implementada, com isso, existe sempre um conjunto de testes que pode ser executado rapidamente.

Deve-se observar que todo processo de *software* tem suas falhas e que muitas organizações de *software* usaram com êxito a XP. O segredo é reconhecer onde um processo pode apresentar fraquezas e adaptá-lo às necessidades específicas de sua organização (PRESSMAN, 2011).

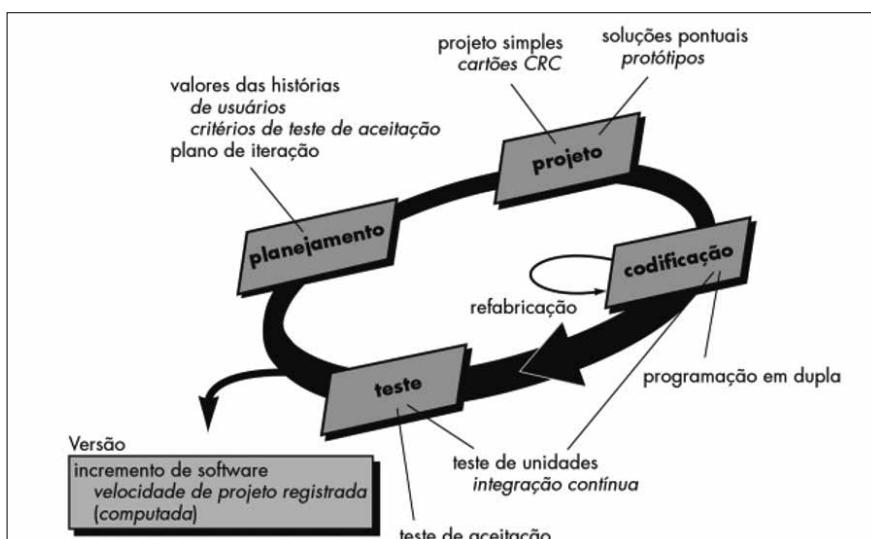
A comunicação no *Extreme Programming* é fundamental, sendo preferível sempre a comunicação pessoal, com clientes e entre os desenvolvedores. A simplicidade busca garantir um *software* simples, com a menor quantidade de classes e métodos, evitando qualquer linha de código desnecessária. A simplicidade também visa garantir que apenas os requisitos necessários venham a ser implementados, evitando requisitos que possam ser utilizados apenas no futuro.

A Programação Extrema valoriza o trabalho em equipe, desenvolvedores, administradores e clientes são todos iguais e todos precisam estar dispostos a ajudar quando necessário. Portanto, sua principal característica é a **PROGRAMAÇÃO EM PARES**.

Baseia-se em cinco princípios fundamentais: comunicação, simplicidade, *feedback*, respeito e coragem e em diversas regras simples, além das já definidas pelo desenvolvimento ágil: o código deve ser escrito usando a técnica de programação em par, todo código deve ter testes unitários, o tempo deve ter um bom espaço para trabalhar, um novo teste será criado quando um *bug* for encontrado, entre outras regras específicas.

Depois de decidido que será utilizado o *Extreme Programming* e preparado um time apto para isso, deve-se gerar um plano de iteração, contendo a história do usuário e qual o objetivo que ele pretende alcançar. Após isso o time se reúne e pensa num projeto simples, pequenas partes que depois de integradas, atenderão as necessidades pontuais do cliente. Com o projeto em mãos, começa a codificação em dupla: dois desenvolvedores sentam lado a lado e concentram-se no código, além de uma técnica de escrever código, ainda precisa de um aprendizado social e isso leva um tempo para ser aperfeiçoado, o desenvolvedor sempre precisa ter em mente que a refatoração do código precisa ser constante, quando algo não é mais usado, deve ser removido, quando parte do código é antiga, deve ser renovada.

FIGURA 66 – CICLO DE ATIVIDADES NO EXTREME PROGRAMMING



FONTE: Ciclo de Atividades no XP. Disponível em: <<http://image.slidesharecdn.com/apresentacaosemtitulo-140123072421-phpapp02/95/xp-extreme-programming-11-638.jpg?cb=1390462029>>. Acesso em: 25 set. 2015.

Após a codificação de pequena parte do projeto, será integrado e testes de aceitação serão realizados sobre ele, e caso seja encontrado algum *bug*, novos testes serão criados para prevenir que aconteça novamente. Após esse ciclo ser completado com sucesso, computa-se o tempo que foi levado e comece uma nova fase, novamente definindo soluções pontuais e colocando em desenvolvimento, repetindo até que o projeto inteiro esteja pronto e funcional.

Conforme destacado por Souza (2015), o *Extreme Programming* utiliza a Orientação ao Objeto como paradigma de desenvolvimento, onde inclui um conjunto de regras e práticas com base nas seguintes atividades: Planejamento, Projeto, Codificação e Teste. No planejamento é realizada a criação de um conjunto de “histórias de usuários” descrevendo as características e funcionalidades requeridas pelo *software* que será construído, estas histórias (semelhantes aos casos de uso) são escritas pelos clientes e colocadas em cartões de indexação, então o cliente atribui uma prioridade a cada história e os desenvolvedores analisam cada história e atribuem um custo a cada uma delas, com base em número de semanas necessárias para o seu desenvolvimento. Se a história precisar de mais de três semanas para desenvolvimento, é solicitado ao cliente que ela seja dividida em histórias menores e por fim com o avanço do projeto, o cliente pode adicionar novas histórias, mudar a sua prioridade, subdividi-la e eliminá-las.



Para maiores informações veja a videoaula sobre o *Extreme Programming*, em que é abordado sobre a programação XP Link: <<https://www.youtube.com/watch?v=AmpCQFx9UbE>>.

3.3 ADAPTATIVE SOFTWARE DEVELOPMENT (ASD)

O *Adaptative Software Development* (ADS) ou Desenvolvimento Adaptativo de *Software* foi proposto por Highsmith para auxiliar no desenvolvimento de sistemas e *softwares* grandes e complexos e concentra-se na colaboração humana e na auto-organização da equipe, com o cliente sempre presente, sendo que o *software* será iterativo e incremental.

Segundo Honorato (2015), o ASD é caracterizado por seis principais propriedades:

- **Orientado a missões:** cada iteração do ciclo é justificada através de uma missão, que pode mudar ao longo do projeto.
- **Baseado em componentes:** o *software* é desenvolvido em pequenas partes.
- **Iterativo:** são desenvolvidos pequenos ciclos, com resultados satisfatórios para cada missão.

- **Prazos pré-fixados:** prazos fixos para a equipe tomar as decisões do projeto o mais cedo possível.
- **Tolerância a mudanças:** as mudanças podem ser frequentes, sempre estando prontos para adaptá-las, com uma constante avaliação de quais componentes podem mudar.

Souza (2015) destaca que a metodologia Desenvolvimento Adaptativo de Software (ASD) é incorporada por três fases:

- **Especulação**
 - ✓ Declara a missão do projeto;
 - ✓ Identifica as restrições do projeto;
 - ✓ Realiza o levantamento dos requisitos básicos.
- **Colaboração**
 - ✓ Filosofia de que pessoas motivadas trabalhando juntas multiplicam seus talentos e resultados.
- **Aprendizado**
 - ✓ Clientes/usuários informam *feedback*;
 - ✓ Revisão dos componentes de *software* desenvolvidos;
 - ✓ Avaliação do desempenho da equipe DAS.

3.4 DYNAMIC SYSTEM DEVELOPMENT METHOD (DSDM)

A metodologia de desenvolvimento de Sistemas Dinâmicos (do inglês *Dynamic Systems Development Method* – DSDM) é uma metodologia de desenvolvimento de *software* originalmente baseada em Desenvolvimento Rápido de Aplicação (RAD). O DSDM é uma metodologia incremental que enfatiza principalmente a participação do usuário final (BUENO, 2015).

Fornece um arcabouço para construir e manter sistemas que satisfazem às restrições de prazo apertadas por meio do uso de prototipagem incremental em um ambiente controlado de projeto que possui nove principais princípios (CS3 CONSULTING SERVICES apud PRESSMAN, 2010):

- O envolvimento do usuário;
- A autonomia do time de desenvolvimento;
- A entrega frequente de produtos funcionais;
- A eficácia das iterações, garantindo a funcionalidade dos incrementos gerados;
- O *feedback* do usuário;
- A reversibilidade do código e suas funcionalidades;
- A previsibilidade do *software* antes do desenvolvimento;
- A ausência de testes no escopo;
- A comunicação entre todos os envolvidos.

A metodologia possui cinco fases definidas em três ciclos iterativos: pré-projeto, ciclo de vida e pós-projeto (PRESSMAN, 2006):

- Estudo de viabilidade: fase onde são estabelecidos os requisitos básicos e as restrições do negócio. Em seguida, é feita uma avaliação se o projeto proposto é viável à metodologia DSDM.
- Estudo do negócio: é definida a arquitetura do *software* a ser construído, bem como os requisitos funcionais e de maior valor ao projeto.
- Iteração do modelo funcional: nesta fase são produzidos protótipos que representam a funcionalidade requerida pelo cliente. Os protótipos têm como objetivo obter requisitos adicionais através do contato com o cliente.
- Iteração de projeto e construção: é executada uma revisão dos protótipos construídos durante a alteração do modelo funcional, para agregar valor aos usuários finais do *software*.
- Implementação: codificam os protótipos como incrementos de *software* funcionais.

3.5 CRYSTAL CLEAR

A família *Crystal Clear* foi desenvolvida por Alistair Cockburn e possui foco na gestão de pessoas, sendo ela pouco definida, possibilitando a adaptação a diversos projetos, focando nas habilidades e talentos de cada pessoa envolvida. Pode-se assim dizer que não há uma única metodologia *Crystal*, e sim várias, de acordo com o projeto e quantidade de pessoas envolvidas. A família *Crystal* possui valores comuns a outras metodologias ágeis, como a comunicação eficaz, a entrega frequente, papéis pré-definidos e equipes com especialistas (SINÉSIO, 2015).

A metodologia *Crystal Clear* está voltada para equipes de dois a oito membros que estejam na mesma sala. Ela não é feita para empresas padronizadas, pois sua metodologia não é completamente especificada, muda de acordo com os pontos fortes e fracos da organização.

De acordo com Sinésio (2015), cada método *Crystal* é caracterizado por uma cor, quatro parâmetros determinam o método de desenvolvimento: tamanho da equipe, localização geográfica, criticidade/segurança e recursos, podendo chegar até 12 casos especiais. *Yellow*: 10 a 20 membros. *Orange*: 20 a 50 membros. *Red*: 50 a 100 membros.

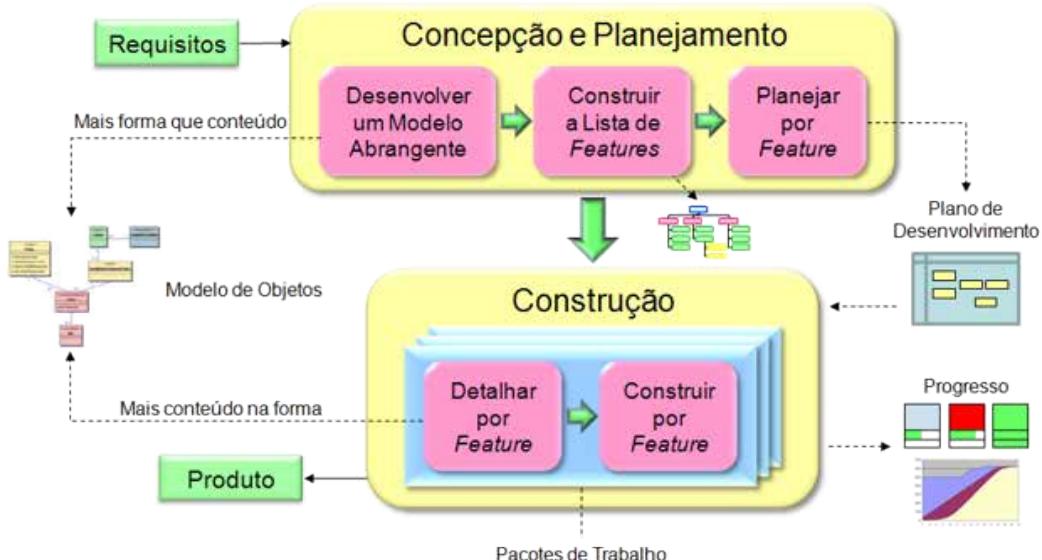
3.6 FEATURE-DRIVEN DEVELOPMENT (FDD)

Em uma equipe que utilizará o FDD ou Desenvolvimento Guiado por Funcionalidade, existem seis funções principais que devem existir: Gerente de projeto, Gerente de desenvolvimento, Arquiteto, Especialista do domínio, Programador Chefe e dono de classe. Diferenciando de outros métodos ágeis, onde o desenvolvimento é feito em conjunto, no FDD, caso alguma alteração precisa ser feita que esteja sob os cuidados de outro dono de classe, mais de um dono devem trabalhar junto para que a alteração seja possível (HEPTAGON, 2015).

A metodologia FDD percorre os seguintes passos:

- O primeiro processo é o desenvolvimento de um modelo abrangente de objetos ou dados. Trata-se da modelagem do problema com o entendimento do domínio do negócio necessário em questão.
- O segundo processo é construir uma lista de funcionalidades, decompondo o modelo em três camadas: áreas de negócio, atividades de negócio, e passos automatizados das atividades.
- A terceira parte é planejar por funcionalidade, abrangendo a estimativa de complexidade das funcionalidades, levando em consideração o valor para o negócio e cliente, e a propriedade.
- A quarta trata-se do detalhamento por funcionalidade, onde dentro de uma iteração em construção, a equipe trabalha para detalhar os requisitos para desenvolvimento e testes. O resultado é um modelo mais detalhado e uma estrutura de código, para ser preenchido posteriormente.
- A quinta é a construção por funcionalidade, onde cada estrutura de código é então preenchida e testada. O resultado é um incremento, que acoplado ao programa principal já poderá ser utilizado pelo cliente.

FIGURA 67 – ESTRUTURA DO FDD



FONTE: Disponível em: <<https://upload.wikimedia.org/wikipedia/commons/2/2f/Fdd.png>>. Acesso em: 26 set. 2015.

Quando aplicado o FDD (*Feature Driven Development*), a equipe precisa estar preparada para trabalhar com funcionalidades como sendo o objeto primário. O primeiro passo é criar um modelo abrangente, modelo que identificará os fundamentos do projeto, que consequentemente sofrerá alterações no decorrer do projeto. Com o modelo base pronto, precisa-se construir uma lista de funcionalidades que deverão ser desenvolvidas, coordenada pelo arquiteto e

identifica os recursos que o *Software* deve ter, comparável às Tarefas no Scrum. Os próximos passos no FDD são conhecidos como “*Design by feature*” e “*Build by feature*”, que é onde acontece a maior parte do projeto, envolvendo *designer*, desenvolvedor, testadores e qualquer outro membro necessário para a conclusão do projeto (HEPTAGON, 2015).



Além do FDD - *Feature-Driven Development* (Desenvolvimento Guiado por Funcionalidade) na área de Gerência de Testes de software encontram-se também:

1. TDD - *Test-Driven Development* (Desenvolvimento Guiado a Testes),
2. DDD - *Domain-Driven Design* (Desenvolvimento Guiado ao Domínio),
3. BDD - *Behavior-Driven Development* (Desenvolvimento Guiado por Comportamento),
4. ATDD - *Acceptance Test-Driven Development* (Desenvolvimento Guiado por Testes de Aceitação).

Informações referentes ao TDD, DDD, BDD e ATDD encontram-se no Tópico 3 desta Unidade 3.

4 BENEFÍCIOS E MALEFÍCIOS DA METODOLOGIA ÁGIL

Conforme resultado das pesquisas bibliográficas referentes à utilização de metodologia ágil, abaixo seguem alguns dos seus principais benefícios:

- O gerenciamento tem o principal papel de eliminar barreiras para o progresso do projeto;
- Foco no projeto e/ou equipe do projeto;
- Equipes que trabalham no mesmo local;
- Benefício em mercados emergentes ou incompreendidos;
- Foco no curto e médio prazo;
- Nasceu em domínios de falhas de baixo custo;
- Pregam o desenvolvimento simultâneo;
- Revisa os processos a cada lição aprendida;
- Disposição para aprendizado buscando aprimorar as capacidades individuais;
- Participação ativa com colaboração, discussão e questionamentos fundamentados no código e nos testes gerados;
- Aprimoramento de técnicas de programação;

Já os resultados negativos identificados nas pesquisas bibliográficas referentes à utilização de metodologia ágil são:

- Não podem ser utilizados em projetos de longo prazo, pois inviabilizam a entrega do *software*;
- A equipe de trabalho deve ter bons conhecimentos na tecnologia aplicada, ou seja, uma equipe sênior;
- Não previne o erro humano, a saída de pessoas chaves, o impacto de pessoas incompetentes, a insubordinação ativa ou passiva, ou a sabotagem deliberada;
- Falta de uma avaliação para atestar empresas com abordagens ágeis;
- Foco excessivo nos prazos e custos inviabiliza o modelo financeiro de longo prazo.

RESUMO DO TÓPICO 2

Neste tópico, você aprendeu que:

- Com as demandas e concorrências subindo, os engenheiros de *software* necessitaram mais uma vez inovar, criando uma metodologia que usasse uma forma ágil de desenvolver, diminuindo os custos e minimizando erros no *software*.
- O manifesto ágil contém quatro principais valores: (1) A capacidade de resposta às mudanças e flexibilidade do *software* acima de um plano pré-estabelecido; (2) A colaboração e participação dos clientes acima das negociações e contratos; (3) Os indivíduos e suas interações acima das ferramentas e procedimentos; (4) O cumprimento dos requisitos e funcionamento do *software* acima de documentação complexa.
- As Metodologias Ágeis de Desenvolvimento de *Software* são recomendadas para projetos em que existem muitas mudanças; os requisitos são passíveis de alterações; a recodificação do programa não acarreta alto custo; a equipe é pequena; as datas de entrega curtas acarretam alto custo; o desenvolvimento rápido é fundamental.
- Kent Beck e mais 16 desenvolvedores, produtores e consultores de *software*, que formavam a Aliança Ágil, assinaram o “Manifesto de Desenvolvimento Ágil” de *Software*, declarando: estamos descobrindo melhores maneiras de se desenvolver *softwares*, fazendo isto e ajudando os outros a fazer isto. Através deste trabalho, nós passamos a valorizar:
 - ✓ Indivíduos e interações mais que processos e ferramentas;
 - ✓ *Software* funcionando mais que documentação abrangente;
 - ✓ Colaboração com o cliente mais que negociação de contratos;
 - ✓ Responder à mudança mais que seguir um plano.
- Entre todos os métodos ágeis as mais importantes são: *Scrum*, *Extreme Programming*, *Adaptative Software Development* (ASD), *Dynamic System Development Method* (DSDM), *Crystal Clear* e *Feature-Driven Development* (FDD).
- O *Scrum* considera uma abordagem mais humana ao solucionar os problemas existentes no desenvolvimento de *Software*. Ao invés de desperdiçar tempo criando documentações extensas e detalhadas que as pessoas acabam não lendo minuciosamente.
- No *Scrum*, as equipes trabalham com *Sprints*. São realizadas reuniões curtas onde o time verifica quais as decisões que devem ser tomadas e os recursos do

product backlog que entram nos *sprints*. Elas também decidem quem trabalha nos *sprints* e quanto tempo dura cada tarefa.

- No *Extreme Programming* (Programação Extrema) a diferença está na forma como o sistema é testado. Não existe especificação do sistema que possa ser usada por uma equipe de teste externa. Para evitar problemas nos testes a abordagem XP enfatiza a importância dos testes do programa, incluindo um foco de testes que reduz as chances de erros não identificados na versão atual do sistema.
- A Programação Extrema valoriza o trabalho em equipe, desenvolvedores, administradores e clientes são todos iguais e todos precisam estar dispostos a ajudar quando necessário. Portanto, sua principal característica é a PROGRAMAÇÃO EM PARES.
- O XP baseia-se em cinco princípios fundamentais: comunicação, simplicidade, *feedback*, respeito e coragem e em diversas regras simples, além das já definidas pelo desenvolvimento ágil: o código deve ser escrito usando a técnica de programação em par, todo código deve ter testes unitários, o tempo deve ter um bom espaço para trabalhar, um novo teste será criado quando um *bug* for encontrado, entre outras regras específicas.
- O *Extreme Programming* utiliza a Orientação ao Objeto como paradigma de desenvolvimento, onde inclui um conjunto de regras e práticas com base nas seguintes atividades: Planejamento, Projeto, Codificação e Teste.
- O *Adaptative Software Development* (ADS) ou Desenvolvimento Adaptativo de *Software* foi proposto por Highsmith para auxiliar no desenvolvimento de sistemas e *softwares* grandes e complexos e concentra-se na colaboração humana e na auto-organização da equipe, com o cliente sempre presente, sendo que o *software* será iterativo e incremental.
- A metodologia (ASD) é incorporada por três fases: Especulação, Colaboração e Aprendizado.
- A metodologia de desenvolvimento de Sistemas Dinâmicos (do inglês *Dynamic Systems Development Method* – DSDM) é uma metodologia de desenvolvimento de *software* originalmente baseada em Desenvolvimento Rápido de Aplicação (RAD). O DSDM é uma metodologia incremental que enfatiza principalmente a participação do usuário final.
- ADS possui cinco fases definidas nos ciclos iterativos pré-projeto, ciclo de vida e pós-projeto: Estudo de viabilidade, Estudo do negócio, Iteração do modelo funcional, Iteração de projeto e construção e Implementação.
- A família *Crystal Clear* foi desenvolvida por Alistair Cockburn e possui foco na gestão de pessoas, sendo ela pouco definida, possibilitando a adaptação a diversos

projetos, focando nas habilidades e talentos de cada pessoa envolvida. Pode-se assim dizer que não há uma única metodologia *Crystal*, e sim várias, de acordo com o projeto e quantidade de pessoas envolvidas.

- O *Crystal* possui valores comuns a outras metodologias ágeis, como a comunicação eficaz, a entrega frequente, papéis pré-definidos e equipes com especialistas.
- No *Feature-Driven Development* (FDD) ou Desenvolvimento Guiado por Funcionalidade existem seis funções principais que devem existir: Gerente de projeto, Gerente de desenvolvimento, Arquiteto, Especialista do domínio, Programador Chefe e dono de classe. Diferenciando de outros métodos ágeis, onde o desenvolvimento é feito em conjunto, no FDD, caso alguma alteração precisa ser feita que esteja sob os cuidados de outro dono de classe, mais de um dono devem trabalhar juntos para que a alteração seja possível.
- Os principais benefícios da metodologia ágil são a colaboração e integração das equipes, foco em projeto de curtos prazos, o desenvolvimento simultâneo e o aprimoramento de técnicas de programação.

AUTOATIVIDADE



1 As Metodologias Ágeis de Desenvolvimento de *Software* são indicadas como sendo uma opção às abordagens tradicionais para desenvolver *softwares*. Analise abaixo sua definição e assinale com V para verdadeiro ou F para falso:

- () Produzem pouca documentação, só é feito o que realmente será útil.
- () A colaboração e participação dos clientes acima das negociações e contratos.
- () Não são recomendadas para projetos que existem muitas mudanças, possui equipe pequena e prazos curtos.
- () Valoriza indivíduos e interações mais que processos e ferramentas.

A sequência correta é:

- a) () F – V – V – V.
- b) () V – V – V – F.
- c) () V – V – F – F.
- d) () V – V – F – V.

2 São princípios dos métodos ágeis:

- a) () Aceitação de mudanças e maior ênfase nos processos em detrimento das pessoas.
- b) () Rejeição de mudanças e envolvimento dos clientes.
- c) () Entrega contínua ao usuário e maior ênfase nas pessoas em detrimento dos processos.
- d) () Maximização da documentação formal e envolvimento dos clientes.

3 São algumas das metodologias de desenvolvimento de *software* consideradas ágeis (*Agile Software Process Models*):

- a) () RUP, XP e DSDM.
- b) () Waterfall, RUP e FDD.
- c) () XP, FDD e RUP.
- d) () Scrum, XP e FDD.

4 A respeito dos princípios do Manifesto ágil, analise abaixo as sentenças e assinale com V para verdadeiro ou F para falso:

- () *Software* em funcionamento é a principal medida de progresso.
- () Mudanças de requisitos, mesmo no fim do desenvolvimento, ainda são bem-vindas.
- () Desenvolvedores e pessoas relacionadas aos negócios devem trabalhar, em conjunto, até o fim do projeto.

- () Garantia da satisfação do consumidor com entrega rápida e contínua de softwares funcionais.
- () Em intervalos regulares, a equipe deve refletir sobre como se tornar mais eficiente.

A sequência correta é:

- a) () V – V – V – V – V.
- b) () V – V – V – F – V.
- c) () V – F – V – F – V.
- d) () V – F – V – F – F.

5 Sobre XP e SCRUM é INCORRETO afirmar:

- a) () No XP, os testes são escritos antes da atividade de desenvolvimento e todas as funcionalidades só possuem valor se forem testadas e obtiverem unanimidade de aprovação.
- b) () O SCRUM tem como características a divisão do processo em pequenos ciclos de desenvolvimento chamados Sprint, o monitoramento do progresso do processo através de reuniões diárias com toda a equipe e, reuniões com os Stakeholders no fim de cada ciclo de desenvolvimento.
- c) () No XP, não há indicação de que é necessário criar documentação no código porém, os documentos tradicionais são reduzidos aos aspectos mais relevantes, visando obter no final do processo, apenas artefatos de grande importância para o projeto.
- d) () SCRUM não especifica a programação em pares ou desenvolvimento orientado a testes, porém especifica a forma de gerenciamento dos requisitos ou características solicitadas.

6 A *Feature Driven Development* (FDD) é uma metodologia ágil de desenvolvimento de software, sobre a qual é correto afirmar:

- a) () Não pode ser combinada a outras técnicas para a produção de sistemas.
- b) () Possui cinco processos: Desenvolver um Modelo Abrangente, Construir a Lista de Funcionalidades, Planejar por Funcionalidade, Detalhar por Funcionalidade e Implementar por Funcionalidade.
- c) () Divide os papéis em dois grupos: papéis chave e papéis de apoio. Dentro de cada categoria, os papéis são atribuídos a um único participante que assume a responsabilidade pelo papel.
- d) () Mantém seu foco apenas na fase de modelagem.

7 A respeito da metodologia Desenvolvimento Adaptativo de Software (ASD) quais são suas três fases:

- a) () Especificação, Cooperação e Aprendizado.
- b) () Especulação, Colaboração e Aprendizado.
- c) () Especulação, Cooperação e Adaptação.
- d) () Especificação, Colaboração e Aprendizado.

8 O DSDM é uma metodologia incremental que enfatiza principalmente a participação do usuário final. A respeito dos seus princípios assinale as sentenças abaixo com V para verdadeiro ou F para falso:

- () O envolvimento do usuário e a autonomia do time de desenvolvimento.
- () O *feedback* só da equipe de desenvolvimento e realização de testes no escopo.
- () A reversibilidade do código e suas funcionalidades e a previsibilidade do *software* antes do desenvolvimento.
- () A comunicação entre todos os envolvidos e a eficácia das iterações.

A sequência correta é:

- a) () V – V – F – V.
- b) () V – V – V – F.
- c) () V – F – V – F.
- d) () V – F – V – V.

9 Cada método Crystal Clear é caracterizado por uma cor e quatro parâmetros determinam o método de desenvolvimento: Tamanho da equipe, Localização geográfica, Criticidade/Segurança e Recursos. A respeito de suas cores assinale a sentença correta:

- a) () *Yellow*: 20 a 50 membros. *Orange*: 10 a 20 membros. *Red*: 50 a 100 membros.
- b) () *Yellow*: 10 a 20 membros. *Orange*: 20 a 50 membros. *Red*: 50 a 100 membros.
- c) () *Yellow*: 50 a 100 membros. *Orange*: 20 a 50 membros. *Red*: 10 a 20 membros.
- d) () *Yellow*: 10 a 20 membros. *Orange*: 50 a 100 membros. *Red*: 20 a 50 membros.

10 A *Feature Driven Development* (FDD) é uma metodologia ágil de desenvolvimento de *software*, sobre a qual é correto afirmar:

- a) () Não pode ser combinada a outras técnicas para a produção de sistemas.
- b) () Possui cinco processos: Desenvolver um Modelo Abrangente, Construir a Lista de Funcionalidades, Planejar por Funcionalidade, Detalhar por Funcionalidade e Implementar por Funcionalidade.
- c) () Divide os papéis em dois grupos: papéis chave e papéis de apoio. Dentro de cada categoria, os papéis são atribuídos a um único participante que assume a responsabilidade pelo papel.
- d) () Mantém seu foco apenas na fase de implementação.

VERIFICAÇÃO, VALIDAÇÃO E TESTES DE SOFTWARE

1 INTRODUÇÃO

O teste é essencial para se garantir a qualidade de *software*. O principal objetivo do teste de *software* é auxiliar na busca de um produto de *software* com o mínimo de erros possível. Para isso, faz-se necessário o uso de ferramentas que auxiliem no processo de teste de *software*, suprindo as diversas necessidades de cada tipo de teste bem como o processo como um todo.

Sommerville (2011) destaca que o teste de *software* serve para evidenciar que o programa faz o que ele realmente deve fazer e para evidenciar os defeitos que existem antes do uso. No processo de teste existem dois objetivos distintos que são demonstrar que o *software* atende seus requisitos e descobrir em que situação o *software* se comporta de forma incorreta.

O teste busca descobrir a maior quantidade de defeitos possível, é importante saber onde os defeitos podem estar. Saber como os defeitos são criados nos dá pistas sobre onde procurá-los durante o teste do sistema (PFLEEEGER, 2004).

Pode-se concluir que determinar a qualidade do *software* pode ter diversos parâmetros a serem analisados de acordo com o entendimento do usuário.

Pressman (2011) afirma que a qualidade de *software* é difícil de definir, porém, é algo que é necessário ser feito e que envolve todas as pessoas (engenheiros de *software*, gerentes, todos os interessados, todos os envolvidos) na gestão de qualidade e as mesmas são responsáveis por ela. Se uma equipe de *software* enfatizar a qualidade em todas as atividades de engenharia de *software*, ela reduzirá a quantidade de reformulações que terá de fazer. Isso resulta em custos menores e mais importante ainda, menor tempo para colocação do produto no mercado. Para obter um *software* de qualidade, devem ocorrer quatro atividades: processo e práticas comprovadas de engenharia de *software*, gerenciamento consistente de projetos, controle global de qualidade e a presença de uma infraestrutura para garantir a qualidade. Para garantir que o trabalho foi realizado corretamente é importante acompanhar a qualidade por meio da verificação dos resultados de todas as atividades de controle de qualidade, medindo a qualidade efetuando

a verificação de erros antes da entrega e de defeitos que acabaram escapando e indo para a produção.

A atividade de teste de *software* possui uma série de limitações, dificuldades e características únicas as quais necessitam serem tratadas durante a sua execução a fim de garantir o seu sucesso. Com isso, planejar e controlar essas atividades se torna fundamental para ter um bom resultado. Definir prazos e gerenciar os riscos se torna imprescindível.

O que quer dizer validação, verificação e teste? É a mesma coisa? Ao realizar uma verificação, a equipe já está realizando a validação e o teste? São coisas bem diferentes, mas que podem ser realizadas pela mesma equipe. Vamos entender um pouco sobre cada um e pelo que eles são responsáveis.

1.1 VALIDAÇÃO

Para entender melhor cada um, para os autores Sommerville (2011) e Pressman (2011) a pergunta da validação é "Fizemos o *software* correto?" Ou seja, a validação é verificar se o *software* tem todos os itens necessários para atender o cliente. O sistema que será entregue ao cliente vai ajudá-lo e o mesmo vai ficar contente.

As empresas fazem dois tipos de validação, uma delas é a baseada nos requisitos. Ou seja, os funcionários vão verificando requisito por requisito, se o mesmo está sendo atendido pelo *software*. Para isso todos os requisitos devem estar muito bem documentados, com bastante detalhes e exemplos.

Já outras empresas preferem realizar os testes e a verificação, e deixar a validação por conta do cliente ou usuário. Algumas encaminham consultores para que a validação seja acompanhada.

1.2 VERIFICAÇÃO

"Fizemos o *software* corretamente?" é o objetivo da etapa de verificação. A verificação fica escondida do usuário final, em comparação à validação. Os encarregados devem buscar e prever erros entre os requisitos. Verificar se todas as etapas de desenvolvimento conforme planejado foram realizadas e da melhor forma.

É verificado se as tecnologias para o desenvolvimento, como banco de dados, a linguagem, as interfaces e etc., foram utilizadas de forma correta. Na verificação um *software* já pode ser utilizado. A verificação não precisa de uma primeira versão do sistema funcionando para realizá-la, diferente das etapas de teste e validação.

1.3 TESTE

"O *software* tem defeito?" é pergunta que assusta qualquer desenvolvedor de sistemas. Mas realizar a fase de teste pode deixar isso mais tranquilo. Um *software* bem testado dificilmente terá erros quando estiver no cliente.

Para realizar os testes é necessário que o *software* esteja pronto, para que os testes sejam montados e executados. Realizar os testes nada mais é que entrar com vários dados de maneira diferente e analisar os dados de saída e seu comportamento. Aqui são realizados vários tipos de teste, como por exemplo, teste de interface, de regras de negócio, de carga, entre outros.

Muitas organizações deixam para montar e realizar os testes apenas quando o produto final estiver pronto. Mas essa etapa leva bastante tempo, e dependendo do resultado, a correção pode trazer mais transtornos para a equipe de desenvolvimento.

Por esse motivo, muitas empresas trabalham com protótipos, onde a cada semana é liberado um para que os testes sejam executados e os novos testes sejam planejados e documentados. Ou seja, a equipe de desenvolvimento liberou um novo protótipo essa semana, a equipe de testes deve criar e documentar os novos testes que irão testar as novas funcionalidades do protótipo. Quando os testes tiverem sido criados, todos os testes criados e os já executados nos protótipos anteriores, devem ser executados novamente, para garantir que o *software* esteja sempre atendendo aos requisitos.

Pode-se encontrar várias definições sobre teste de *software*, entre elas destaca-se que é o processo que visa sua execução de forma controlada, com o objetivo de avaliar o seu comportamento baseado no que foi especificado. A execução dos testes é considerada um tipo de validação. (RIOS; MOREIRA, 2013).

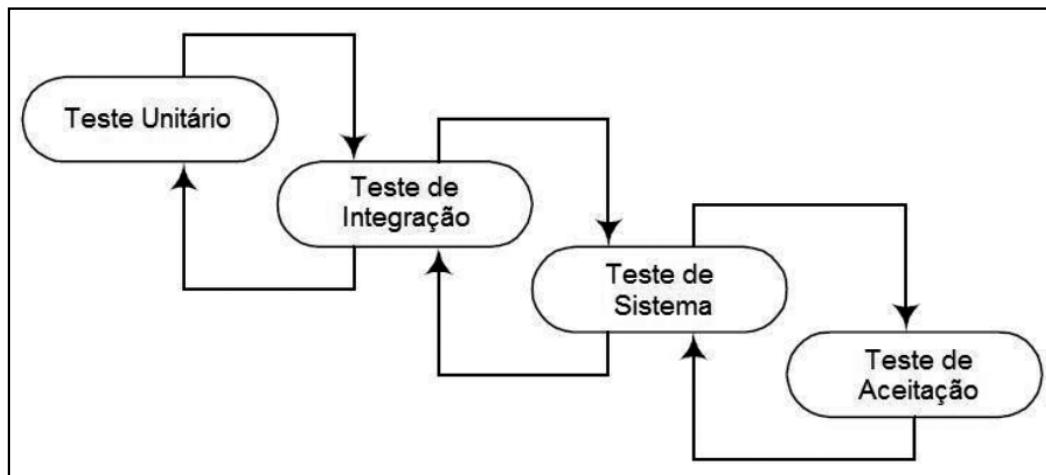
Na área de testes também existem diversos tipos de teste que são aplicados em estágios diferentes. Conforme Rios e Moreira (2013):

- **Teste Caixa Preta (*Black Box*):** visa verificar a funcionalidade e a aderência aos requisitos, em uma ótica externa ou do usuário, sem se basear em qualquer conhecimento do código e da lógica interna do componente testado.
- **Teste Caixa Branca (*White Box*):** visa avaliar as cláusulas de código, a lógica interna do componente codificado, as configurações e outros elementos técnicos.

Pfleeger (2004) afirma que muitos tipos de testes são realizados antes da entrega do sistema para o cliente, alguns testes dependem do que está sendo testado, outros do que se pretende saber.

Os testes de *software* são executados em diferentes níveis (ou estágios) do desenvolvimento de um *software*.

FIGURA 68 – OS QUATRO PRINCIPAIS NÍVEIS TÍPICOS DE TESTE DE SOFTWARE



FONTE: Adaptado de: Craig (2002)

A figura acima, adaptada de Craig (2002), ilustra os quatro níveis de teste em relação ao escopo do *software* sob teste e a realidade de ambiente existente. O autor afirma que “cada nível é definido por um ambiente particular, que pode incluir configuração de *hardware*, configuração de software, interfaces, testadores etc. Note que na medida em que você se move para níveis mais altos de teste o ambiente se torna mais realístico”.

Em literaturas encontram-se alguns tipos de testes conforme descrito nos níveis a seguir:

- **Teste de Unidade:** o teste é realizado em cada componente do programa isoladamente, no qual se verifica se ele funciona de forma adequada aos tipos de entradas esperadas. Normalmente, esse tipo de teste é realizado em um ambiente controlado. Além disso, a equipe verifica as estruturas de dados internas, a lógica e as condições limite para os dados de entrada e saída (PFLEEGER, 2004).
- **Teste de Integração:** tem o objetivo de provocar falhas associadas às interfaces entre os módulos quando esses são integrados para construir a estrutura do *software* que foi estabelecida na fase de projeto (DIAS NETO, 2015).
- **Teste de Sistema:** Dias Neto (2015) descreve que o teste de sistema avalia o *software* em busca de falhas utilizando o mesmo como se fosse um usuário final. Sendo assim os testes são com as mesmas condições e com os mesmos dados de entrada que um usuário utilizaria. Rios e Moreira (2013) acrescentam que é nesse estágio que são realizados os testes de carga, *performance*, usabilidade, compatibilidade, segurança e recuperação.

- **Teste de Aceitação:** é realizado em conjunto com os clientes e nele o sistema é verificado em comparação com a descrição dos requisitos do cliente (PFLEEGER, 2004).

Existem diversos outros tipos de testes que podem ser executados no processo de desenvolvimento se adequando à realidade da empresa. Porém, os testes automatizados têm ganhado bastante notoriedade, conforme afirma Sommerville (2011, p. 147):

[...] o uso dos testes automatizados tem aumentado consideravelmente nos últimos anos. Entretanto, os testes nunca poderão ser totalmente automatizados, já que testes automáticos só podem verificar se um programa faz aquilo a que é proposto. É praticamente impossível usar testes automatizados para testar os sistemas que dependem de como as coisas estão (por exemplo, uma interface gráfica de usuário), ou para testar se um programa não tem efeitos colaterais indesejados.

Partindo desse pressuposto, verifica-se que o ideal é fazer uma junção das técnicas existentes se adequando ao processo de desenvolvimento de *software*.

2 EQUIPE DE TESTE

Não basta ter um *software* para realizar teste de alta qualidade e não ter uma equipe treinada para utilizar o *software* de teste. Montar essa equipe é um dos maiores problemas encontrados pelas empresas, pois os membros não podem ser apenas programadores, deve também ter membros que entendam as regras de negócio, onde o *software* vai trabalhar. Ou seja, funcionários que tenham a mesma lógica do usuário final. Desta forma, fica mais fácil identificar os erros do sistema.

Também existe o contrário, onde as empresas montam uma equipe de teste apenas com profissionais para a área onde o *software* foi desenvolvido. Desta forma acontece o que o autor Ramos (2010) descreveu.

Fazem aquele teste de usuário, mesmo assim superficial, não fazem teste do banco de dados, do desempenho do sistema, de análise do código, não catalogam os defeitos em níveis de defeitos, não criam regras de mensagens de erros para ajudar o usuário a contornar os possíveis erros etc. Hoje montar uma equipe de testes é importante, se estuda muito teste de *software*, são raros os profissionais especializados e são muito bem remunerados. (RAMOS, 2010).

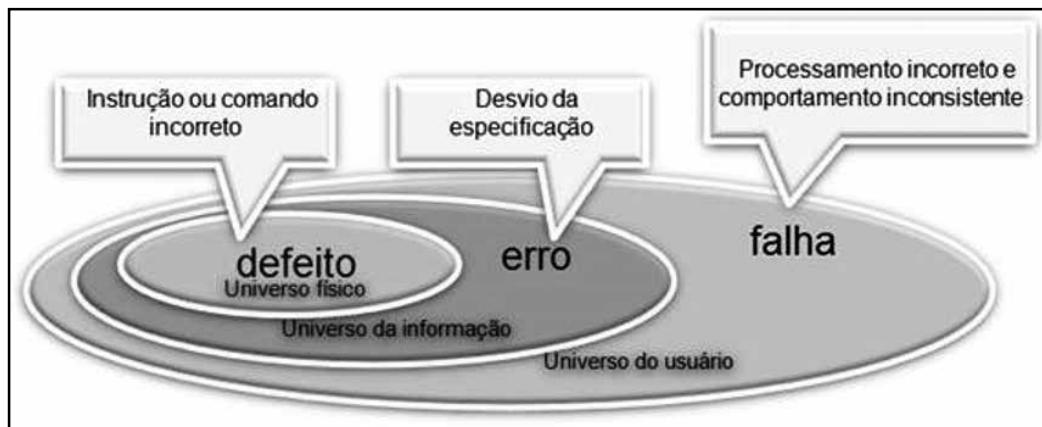
Montar uma equipe não é fácil, a mesma deve ter profissionais da área para onde o *software* está sendo desenvolvido, e também profissionais da área de tecnologia. Desta forma a empresa terá testes de negócio e também de *performance*.

3 ERROS DE SOFTWARE

Continuando nossa discussão sobre teste de *software*, se abordará um pouco mais sobre erros de *software*, e para isso, se irá conceituar melhor a diferença entre defeito, erro e falha. As definições que iremos usar aqui seguem a terminologia padrão para Engenharia de *Software* do IEEE – *Institute of Electrical and Electronics Engineers* (NETO, 2015 apud IEEE 610, 1990).

- Defeito é um ato inconsistente cometido por um indivíduo ao tentar entender uma determinada informação, resolver um problema ou utilizar um método ou uma ferramenta. Por exemplo, uma instrução ou comando incorreto.
- Erro é uma manifestação concreta de um defeito num artefato de *software*. Diferença entre valor obtido e valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução de um programa constitui erro.
- Falha é um comportamento operacional do *software* diferente do esperado pelo usuário. Uma falha pode ter sido causada por diversos erros e alguns erros podem causar uma falha.

FIGURA 69 – DIFERENÇA DEFEITO X ERRO X FALHA



FONTE: Disponível em: <[http://www.devmedia.com.br/artigo-engenharia-de-software-introdução-a-teste-de-software/8035](http://www.devmedia.com.br/artigo-engenharia-de-software-introducao-a-teste-de-software/8035)>. Acessado em: 26 set. 2015.

Segundo Molinari (2003), defeito é uma não conformidade com requisitos especificados. Pode ser qualquer elemento de requisito, desenhado ou implementado que se não for alterado, pode causar um desenho, implementação, teste, uso ou manutenção impróprio do produto. *Bug* pode ser qualquer problema, que pode não ser um defeito.

Nem todas as falhas são consideradas realmente um *bug*, pois pode ser um caso de má utilização do *software* por parte do usuário. Para que um *bug* ocorra, de acordo com Molinari (2003), basta que uma ou mais das quatro regras da indústria de *software* apresentadas a seguir ocorram simultaneamente:

- Software não faz algo que a especificação do produto diz que faz;
- Software faz algo que na especificação diz que não é para fazer;
- Software faz algo que a especificação do produto não menciona;
- Software é difícil de entender, difícil de usar, lento, ou simplesmente aos olhos do testador “o usuário não gostará”.

Concertar um erro “na hora” em que o mesmo é identificado auxilia num mecanismo de controle pelo fato de que quanto mais cedo o programador aprender com seus erros, menos chances têm de repeti-los (MAGUIRE, 1994).

Um *software* deve ter todos os seus elementos testados, pois não há maneira de prever ou identificar onde estão os erros, sem que sejam feitos diferentes tipos de verificações. Outra questão é que quanto mais tarde for identificado um erro, maior é o seu custo.

4 TIPOS DE TESTE

De acordo com Oliveira (2007), a *Rational Unified Process* classifica testes sob os cinco fatores de qualidade do modelo FURPS: *Functionality* (Funcionalidade), *Usability* (Usabilidade), *Reliability* (Confiabilidade), *Performance* (Desempenho) e *Supportability* (Suportabilidade). Para cada um desses fatores, ou dimensões de qualidade, como denomina o referido processo, existe um ou mais tipos de teste associado:

TABELA 22 – FURPS X TIPOS DE TESTE

Dimensão de qualidade	Tipos de teste
Funcionalidade	Teste funcional Teste de segurança Teste de volume
Usabilidade	Teste de usabilidade
Confiabilidade	Teste de integridade Teste de estrutura Teste de <i>stress</i>
Desempenho	Teste de avaliação de desempenho Teste de contenção Teste de carga Teste de perfil de desempenho
Suportabilidade	Teste de configuração Teste de instalação

FONTE: O autor

4.1 FUNCIONALIDADE

- Teste funcional: define teste funcional como um processo de tentar encontrar discrepâncias entre o programa e sua especificação externa;
- Teste de segurança: destinado a garantir que o objetivo do teste e os dados (ou sistemas) possam ser acessados apenas por determinados atores;
- Teste de volume: verifica o comportamento do *software* quando submetido a grandes volumes de dados (OLIVEIRA, 2007).

4.2 USABILIDADE

- Teste de usabilidade: corresponde à facilidade de uso do *software* (OLIVEIRA, 2007).

4.3 CONFIABILIDADE

- Teste de integridade: destinado a avaliar a resistência do *software* a falhas e a compatibilidade técnica em relação à linguagem, sintaxe e utilização de recursos;
- Teste de estrutura: é utilizado em dois contextos principais. O primeiro, está relacionado a testar a estrutura de itens internos do código à procura de pontos fracos e erros estruturais. O segundo está relacionado a testar estruturas de sites da web para verificar se todos os *links* (estáticos ou ativos) estão conectados corretamente;
- Teste de stress: objetiva avaliar o comportamento do sistema em condições anormais, com cargas de trabalho extremas, memória insuficiente, hardware e serviços indisponíveis, ou recursos compartilhados limitados (OLIVEIRA, 2007).

4.4 DESEMPENHO

- Teste de avaliação e desempenho: procura avaliar o desempenho do *software* mediante uma carga de trabalho conhecida;
- Teste de contenção: tem o objetivo de avaliar se o *software* lida de forma aceitável quando recursos compartilhados recebem demandas de diferentes atores;
- Teste de carga: visa avaliar os limites operacionais de um sistema mediante a carga de trabalho variável;

- Teste de perfil de desempenho: visa identificar gargalos de desempenho, a partir do monitoramento do *software*, incluindo fluxo de execução, acesso a dados e chamada de funções (OLIVEIRA, 2007).

4.5 SUPORTABILIDADE

- Teste de configuração: tem o propósito de garantir que o *software* funcione de maneira aceitável quando submetido a diferentes configurações de *hardware* ou *software*;
- Teste de instalação: visa garantir que o sistema seja instalado conforme o esperado, mesmo em condições anormais, como espaço insuficiente em disco e interrupção de energia (OLIVEIRA, 2007).

5 PROCESSO DE TESTE DE SOFTWARE

Para avaliar um processo de teste de *software* adequado, leva-se em consideração diversos fatores incluindo o porte da empresa e a sua realidade. Baseado nesse fato surge novos olhares para as metodologias ágeis que trazem uma forma alternativa no desenvolvimento de *software*. Essas metodologias têm por objetivo orientar o processo para se adequar a um processo mais dinâmico e eficiente.

Normalmente o processo de testes deve ser baseado em metodologia aderente ao processo de desenvolvimento, em pessoal técnico qualificado, em ambiente e ferramentas adequadas (RIOS; MOREIRA, 2013).

Pfleeger (2004) afirma que as etapas do processo devem ser planejadas, e que o processo de teste tem vida própria no ciclo de desenvolvimento e o mesmo pode ser realizado paralelamente com outras atividades de desenvolvimento.

Rios e Moreira (2013) caracterizam o processo de testes em execução das principais etapas e dos seus desdobramentos (subetapas), as quais são o Planejamento; Procedimentos iniciais; Preparação; Especificação; Execução; Entrega.

Baseado nesses conceitos percebe-se que o processo deve ser mais dinâmico e flexível. A monitoração deve ser constante para acompanhar a evolução dos resultados e sugerindo modificação quando necessária.

De acordo com Sommerville (2011), as abordagens de desenvolvimento nas metodologias ágeis levam em consideração o projeto e a implementação como sendo atividades centrais no processo de *software*. Eles incorporaram outras atividades, como elicitação de requisitos e testes no projeto e na implementação.

5.1 PRÁTICAS DE DESENVOLVIMENTO

5.1.1 TDD - *Test-Driven Development* (Desenvolvimento Guiado a Testes)

Desenvolvimento Guiado por Teste é aquele que se escreve primeiramente os testes para posteriormente escrever o código. O TDD é parte do processo de desenvolvimento ágil, utilizado em metodologias como o XP (Programação Extrema) e sendo uma das técnicas que auxiliam na melhoria de qualidade do processo de desenvolvimento. O TDD torna mais eficiente o processo (ROCHA, 2015).

Pires (2015) ressalta que o processo de desenvolvimento do TDD aborda os parâmetros *Red*, *Green* e *Refactor*:

1. Escrever um teste, mesmo sem ter escrito o código real a ser testado;
2. Executar os testes e acompanhar a falha (*Red*);
3. Escrever a funcionalidade do sistema que irá ser testada;
4. Testar novamente, agora para passar (*Green*);
5. Refatorar a funcionalidade e escrever por completo (*Refactor*);
6. Próxima estória ou caso de uso e iniciar novo teste.

O TDD é um conjunto de técnicas que culminam em um teste de ponta a ponta (ROCHA, 2015).

5.1.2 DDD - *Domain-Driven Design* (Desenvolvimento Guiado ao Domínio)

Desenvolvimento Guiado ao Domínio são padrões e princípios que ajudam em seus esforços para construir aplicações que refletem uma compreensão e a satisfação das exigências do seu negócio. Trata da modelagem do domínio real por primeiramente entendê-la completamente e então colocar todas as terminologias, regras, e lógica em uma representação abstrata dentro do seu código, tipicamente em forma de um modelo de domínio. DDD não é um *framework*, mas ele tem um conjunto de blocos ou conceitos que podem ser incorporados em sua solução (SCHIASSATO; PEREIRA, 2015).

O foco é no Domínio do *Software*, no propósito que o *software* deve atender, é a automatização de um processo de negócio. O DDD traz abordagens de como fazer isto, como atender um domínio complexo de informações. Qualquer abordagem de DDD é muito bem aceita numa metodologia ágil (PIRES, 2015).

Pires (2015) ainda afirma que o DDD é importante em um sistema complexo e não é aconselhável para um sistema simples. Na maioria dos sistemas corporativos são encontradas diversas regras de negócio e cada uma com sua particularidade e complexidade. Iniciar um projeto usando a abordagem de DDD previne que o sistema cresça cada vez mais de uma forma não orientada ao domínio.

5.1.3 BDD – *Behavior-Driven Development* (Desenvolvimento Guiado Por Comportamento)

BDD é uma evolução do TDD, de forma explícita, BDD relaciona “*Test-Driven Development*” com “*Domain-driven Design*” tornando a relação entre essas duas abordagens consideravelmente mais evidente. BDD colabora para que o desenvolvimento foque na entrega de valor, através da formação de um vocabulário comum, reduzindo a distância entre negócio e tecnologia (ELEMAR JR., 2015).

BDD se destina a satisfazer as necessidades de ambos os usuários (técnicos e de negócio), BDD pode ser realizado utilizando *frameworks* de testes de unidade, ou com *frameworks* específicos de BDD que têm surgido em diversas linguagens (SCHIASSATO; PEREIRA, 2015).

BDD associa os benefícios de uma documentação formal, escrita e mantida pelo negócio, com testes de unidade que demonstram que essa documentação é efetivamente válida. Na prática, isso garante que a documentação deixa de ser um registro estático, que se converte em algo gradualmente ultrapassado, em um artefato vivo que reflete constantemente o estado atual de um projeto (ELEMAR JR., 2015).

5.1.4 ATDD - *Acceptance Test-Driven Development* (Desenvolvimento Guiado por Testes de Aceitação)

As equipes *Scrum* aprenderam a diminuir o fluxo de trabalho que passa por uma *sprint* executando o desenvolvimento baseado em teste de aceitação. No ATDD, o trabalho ocorre em resposta a testes de aceitação. O ATDD pode ser considerado como análogo ao TDD (GRIEBLER, 2015).

5.1.5 FDD - *Feature Driven Development* (Desenvolvimento Guiado por Funcionalidades)

O FDD já foi apresentado no Tópico 3, mas serve para gerenciar e desenvolver projetos de software através de um conjunto de atividades simplificadas, de maneira a estimular o compartilhamento do conhecimento acerca do *software* e da criação de bons códigos, permitindo então que o principal objetivo da FDD: “resultados frequentes, tangíveis e funcionais”, seja alcançado num projeto de desenvolvimento de *software* (GRIEBLER, 2015).

6 FERRAMENTAS PARA AUTOMAÇÃO DE TESTE

A escolha das ferramentas levou em conta sua popularidade no mercado. Em particular, atualmente a Mercury, comprada pela HP, é a líder com cerca de 50% do mercado mundial de ferramentas de teste de *software*. Segue abaixo quadro com as ferramentas para teste de *software* mais utilizado.

TABELA 23 – FERRAMENTAS DE TESTE DE SOFTWARE

Tipo de ferramenta: rastreabilidade de Erros		
Ferramenta	Fabricante	Descrição (segundo fabricante)
TestDirector for Quality Center (versão 8.2)	Mercury/HP	Ferramenta de gerenciamento de teste e rastreabilidade de informações de um projeto de <i>software</i> , integrada e centralizada. Algumas funcionalidades e características: processo repetível e consistente de manutenção e gerenciamento de requisitos; planejamento, controle e execução de testes; análise de resultados e gerenciamento de defeitos e problemas; rastreabilidade de informações envolvidas no processo.
Bugzilla (versão 3.0.2)	Bugzilla	Bugzilla é uma ferramenta que auxilia no desenvolvimento de <i>software</i> , oferecendo rastreabilidade de erros. Novas funcionalidades da última versão: estrutura de banco de dados otimizada para aumento de <i>performance</i> e escalabilidade; recursos de segurança avançados; ferramenta de consulta avançada que “lembra” o usuário sobre últimas buscas; capacidades integradas de <i>e-mail</i> ; personalização de usuários e preferências de <i>e-mail</i> ; sistema de permissões; utilizada como ferramenta de rastreabilidade de erros do navegador Mozilla.
ClearQuest (versão 7.0)	Rational/IBM	Ferramenta de rastreabilidade de erros. Algumas funcionalidades e características: permite automatização de processos; geração de relatórios; permite um fluxo flexível de gerenciamento de erros e mudanças através do ciclo de vida da aplicação; proporciona uma melhor visualização de informações, previsão e controle.

Bugzero (versão 5.1)	Websina	Ferramenta de rastreabilidade de erros. Algumas funcionalidades e características: rastreabilidade de erros, problemas, mudanças, casos de teste; gerenciamento de caso de uso; pode ser utilizada como centro de informações de dados de cliente, solicitações e apoio a suporte; configurações flexíveis e avançadas.
Tipo de ferramenta: Teste Funcional		
Ferramenta	Fabricante	Descrição
TestDirector for Quality Center (versão 9.0)	Mercury/HP	Ver acima
QuickTest Professional (versão 0.1)	Mercury/HP	Ferramenta para testes funcionais e de regressão automatizados, com uso de gravação de objetos pela interface do sistema a ser testado. Algumas funcionalidades e características: interação com outras ferramentas do fabricante; pré-configurações para ganho de tempo na criação de <i>scripts</i> ; reconhecimento avançado de objetos de interface; relatório passo a passo com recurso de vídeo.
WET Web Tester (versão 1.0.0)	Qantom	Ferramenta <i>open source</i> de testes automatizados para a <i>web</i> . Algumas funcionalidades e características: desenvolvimento de teste usando interface gráfica; extensibilidade com relação aos scripts; repositório de objetos para manutenção de scripts; identificação de objetos usando múltiplos parâmetros.
TestPartner (versão 6.0)	Compuware	Ferramenta de automatização de testes através de interface de usuário. Algumas funcionalidades e características: gravação e reprodução; repositório de banco de dados, mapeamento de objetos e imagens; checagens avançadas; suporte a múltiplos ambientes; funções de scripts modulares.
Tipo de ferramenta: Teste de Performance		
Ferramenta	Fabricante	Descrição
LoadRunner (versão 8.1)	Mercury/HP	Ferramenta que previne problemas de desempenho em ambiente de produção detectando gargalos antes da instalação ou upgrade de um novo sistema. Algumas funcionalidades e características: cargas de dados conforme necessário; suporte a ambiente; suporte de monitoramento e diagnósticos compatível com várias tecnologias; monitoramento de rede, aplicações e servidores para medir performance de cada camada, servidor e componentes e rastrear gargalos.
Web Performance Suite (versão 3.4)	Web Performance Inc.	Ferramenta para teste de performance para web, teste de carga e teste e estresse. Algumas funcionalidades e características: inclui um módulo de análise cliente-side; módulo de teste de carga de dados e módulo de análise de servidor; compatível com várias tecnologias.

Performance Tester (versão 6.1)	Rational/IBM	Ferramenta de teste multiusuário para validar escalabilidade de uma aplicação antes da instalação em produção. Algumas funcionalidades e características: cria, executa e analisa testes para verificar confiabilidade das aplicações; extensões para outras tecnologias; identificação e suporte automático para respostas de servidores dinâmicos; compatível com Linux e Windows.
---------------------------------	--------------	--

FONTE: O autor

RESUMO DO TÓPICO 3

Neste tópico, você aprendeu que:

- O teste é essencial para se garantir a qualidade de *software*. O principal objetivo do teste de *software* é auxiliar na busca de um produto de *software* com o mínimo de erros possível.
- O teste de *software* serve para evidenciar que o programa faz o que ele realmente deve fazer e para evidenciar os defeitos que existem antes do uso. No processo de teste existem dois objetivos distintos que é demonstrar que o *software* atende seus requisitos e descobrir em que situação o *software* se comporta de forma incorreta.
- A validação é verificar se o *software* tem todos os itens necessários para atender ao cliente. O sistema que será entregue ao cliente vai ajudá-lo e o mesmo vai ficar contente. Para a validação a pergunta é “Fizemos o *software* correto?”
- A verificação fica escondida do usuário final, em comparação à validação. Os encarregados devem buscar e prever erros entre os requisitos. Verificar se todas as etapas de desenvolvimento conforme planejado foram realizadas e da melhor forma. Para a verificação a pergunta é “Fizemos o *software* corretamente?”.
- Realizar os testes nada mais é que entrar com vários dados de maneira diferente e analisar os dados de saída e seu comportamento. Aqui são realizados vários tipos de teste, como por exemplo, teste de interface, de regras de negócio e de carga. Para testar a pergunta é “O *software* tem defeito?”.
- Teste de *software* é o processo que visa sua execução de forma controlada, com o objetivo de avaliar o seu comportamento baseado no que foi especificado. A execução dos testes é considerada um tipo de validação.
- Na área de testes, os tipos de Teste Caixa Preta, que visam verificar a funcionalidade e a aderência aos requisitos, em uma ótica externa ou do usuário, sem se basear em qualquer conhecimento do código e da lógica interna do componente testado e o de Teste Caixa, que visam avaliar as cláusulas de código, a lógica interna do componente codificado, as configurações e outros elementos técnicos.
- Os tipos de testes conforme seus níveis são:
 - ✓ Teste de Unidade: o teste é realizado em cada componente do programa isoladamente, no qual se verifica se ele funciona de forma adequada aos tipos de entradas esperadas.

- ✓ Teste de Integração: tem o objetivo de provocar falhas associadas às interfaces entre os módulos quando esses são integrados para construir a estrutura do *software* que foi estabelecida na fase de projeto.
- ✓ Teste de Sistema: avalia o *software* em busca de falhas utilizando o mesmo, como se fosse um usuário final.
- ✓ Teste de Aceitação: é realizado em conjunto com os clientes e nele o sistema é verificado em comparação com a descrição dos requisitos do cliente.
- Continuando nossa discussão sobre teste de *software*, se abordará um pouco mais sobre erros de *software*, e para isso, se irá conceituar melhor a diferença entre defeito, erro e falha:
 - ✓ Defeito é um ato inconsistente cometido por um indivíduo ao tentar entender uma determinada informação, resolver um problema ou utilizar um método ou uma ferramenta. Por exemplo, uma instrução ou comando incorreto.
 - ✓ Erro é uma manifestação concreta de um defeito num artefato de *software*. Diferença entre valor obtido e valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução de um programa constitui erro.
 - ✓ Falha é um comportamento operacional do *software* diferente do esperado pelo usuário. Uma falha pode ter sido causada por diversos erros e alguns erros podem causar uma falha.
- A *Rational Unified Process* classifica testes sob os cinco fatores de qualidade do modelo FURPS: *Functionality* (Funcionalidade), *Usability* (Usabilidade), *Reliability* (Confiabilidade), *Performance* (Desempenho) e *Supportability* (Suportabilidade). Para cada um desses fatores, ou dimensões de qualidade, como denomina o referido processo, existe um ou mais tipos de teste associado:
 - ✓ Funcionalidade: Teste funcional, Teste de segurança e Teste de volume.
 - ✓ Usabilidade: Teste de usabilidade.
 - ✓ Confiabilidade: Teste de integridade, Teste de estrutura e Teste de *stress*.
 - ✓ Desempenho: Teste de avaliação de desempenho, Teste de contenção, Teste de carga e Teste de perfil de desempenho.
 - ✓ Suportabilidade: Teste de configuração e Teste de instalação.
- As práticas de desenvolvimento na área de testes são:
 - ✓ TDD – *Test - Driven Development*: o Desenvolvimento Guiado a Testes; se escreve primeiramente os testes para posteriormente escrever o código, o processo aborda os parâmetros *Red*, *Green* e *Refactor*: (1) Escrever um teste, mesmo sem ter escrito o código real a ser testado; (2) Executar os testes e acompanhar a falha (*Red*); (3) Escrever a funcionalidade do sistema que irá ser testada; (4) Testar novamente, agora para passar (*Green*); (5) Refatorar a funcionalidade e escrever por completo (*Refactor*); (6) Próxima estória ou caso de uso e iniciar novo teste.

- ✓ DDD – *DOMAIN - DRIVEN DESIGN*: no desenvolvimento guiado ao domínio o foco é no Domínio do *Software*, no propósito que o *software* deve atender, é a automatização de um processo de negócio. O DDD traz abordagens de como fazer isto, como atender um domínio complexo de informações. Qualquer abordagem de DDD é muito bem aceita numa metodologia ágil.
 - ✓ BDD – *Behavior - Driven Development*: o desenvolvimento guiado por comportamento associa os benefícios de uma documentação formal, escrita e mantida pelo negócio, com testes de unidade que demonstram que essa documentação é efetivamente válida. Na prática, isso garante que a documentação deixa de ser um registro estático, que se converte em algo gradualmente ultrapassado, em um artefato vivo que reflete constantemente o estado atual de um projeto.
 - ✓ ATDD-*ACCEPTANCE TEST-DRIVEN DEVELOPMENT*: o desenvolvimento guiado por testes de aceitação; o trabalho ocorre em resposta a testes de aceitação. O ATDD pode ser considerado como análogo ao TDD.
 - ✓ FDD - *FEATURE DRIVEN DEVELOPMENT*: desenvolvimento guiado por funcionalidades serve para gerenciar e desenvolver projetos de *software* através de um conjunto de atividades simplificadas, de maneira a estimular o compartilhamento do conhecimento acerca do *software* e da criação de bons códigos.
- As ferramentas para teste de *software* mais utilizadas são:
 - ✓ Tipo de ferramenta para rastreabilidade de erros: *TestDirector for Quality Center*, *Bugzilla*, *ClearQuest* e *Bugzero*.
 - ✓ Tipo de ferramenta para Teste Funcional: *TestDirector for Quality Center*, *QuickTest Professional*, *WET Web Tester* e *TestPartner*.
 - ✓ Tipo de ferramenta para Teste de *Performance*: *LoadRunner*, *Web Performance Suite* e *Performance Tester*.

AUTOATIVIDADE



- 1 O principal objetivo do processo de verificação e validação (V&V) de *software* é estabelecer confiança de que o sistema de *software* atende tanto a sua especificação quanto às expectativas de seus usuários finais. Além das atividades de inspeção de *software*, outras atividades de suma importância no contexto do processo de V&V são aquelas relacionadas
- a) () aos testes de *software*.
 - b) () à manutenção de *software*.
 - c) () à estimativa de custo de *software*.
 - d) () ao gerenciamento de configuração de *software*.
- 2 A fase de elaboração dos testes de *software* é uma das partes mais importantes, no desenvolvimento de um *software*. Sobre o teste de caixa branca, assinale a alternativa correta.
- a) () Teste feito pela equipe de testadores de *software*.
 - b) () Teste executado pelo usuário final do *software*.
 - c) () Teste executado, após a implantação do *software*.
 - d) () Teste feito pelo próprio programador que verifica se o código que foi construído é funcional.
- 3 Dentro as técnicas de teste de *software*, há os testes denominados de caixa preta e aqueles denominados de caixa branca. Testes do tipo caixa
- a) () preta visam exercitar as interfaces do *software* sob teste.
 - b) () preta e branca visam detectar os mesmos tipos de erros existentes no *software* sob teste.
 - c) () preta não são aplicáveis a *software* de pequeno porte.
 - d) () branca também são chamados de testes comportamentais.
- 4 O processo de confirmação que um *software* vai ao:
- a) () Validação.
 - b) () Verificação.
 - c) () Precisão.
 - d) () Confiabilidade.
- 5 Qual é o tipo de teste que focaliza o esforço de verificação na menor unidade de projeto de *software*, isto é, no componente ou no módulo de *software*?
- a) () Teste de integração.
 - b) () Teste de unidade.
 - c) () Teste de validação.
 - d) () Teste de sistema.

- 6 Os testes de *software* são executados em diferentes níveis do desenvolvimento de um *software*. A respeito dos quatro principais níveis de testes de *software*, assinale abaixo com V para verdadeiro ou F para falso:
- () Teste de Unidade é realizado em conjunto com os clientes e nele o sistema é verificado em comparação com a descrição dos requisitos do cliente.
 - () O teste de Integração tem o objetivo de provocar falhas associadas às interfaces entre os módulos quando esses são integrados para construir a estrutura do *software* que foi estabelecida na fase de projeto.
 - () O teste de Sistema avalia o *software* em busca de falhas utilizando o mesmo, como se fosse um usuário final.
 - () O teste de Aceitação é realizado em cada componente do programa isoladamente, no qual se verifica se ele funciona de forma adequada aos tipos de entradas esperadas.
- A sequência correta é:
- a) () F – F – F – V.
 - b) () V – V – V – F.
 - c) () F – V – F – F.
 - d) () F – V – V – F.
- 7 Quais são os cinco fatores de qualidade classificados por *Rational Unified Process* nos testes de *software*:
- a) () Funcionalidade, Acessibilidade, Confiabilidade, Performance e Conectividade.
 - b) () Requisitos, Usabilidade, Segurança, Desempenho e Suportabilidade.
 - c) () Requisitos, Acessibilidade, Segurança, Performance e Conectividade.
 - d) () Funcionalidade, Usabilidade, Confiabilidade, Desempenho e Suportabilidade.
- 8 O desenvolvimento dirigido a testes (TDD, do Inglês *Test-Driven Development*) é uma abordagem de desenvolvimento de *software* na qual se intercalam testes e desenvolvimento de código. Uma das características da abordagem TDD é:
- a) () A redução da importância da automatização dos testes.
 - b) () O maior custo associado aos testes de regressão.
 - c) () A sua utilidade no desenvolvimento de *softwares* novos.
 - d) () A sua adequação a processos de *software* sequenciais.

GOVERNANÇA DE TECNOLOGIA DA INFORMAÇÃO

1 INTRODUÇÃO

A Tecnologia da Informação e da Comunicação (TIC) já faz parte da gestão operacional e estratégica das organizações, é um elemento fundamental na direção do seu progresso, visto que cada vez mais, as suas operações estão dependentes de algum tipo de tecnologia que pode ser empregado na melhoria de processos de negócio, com o intuito de proporcionar maior agilidade e fornecer as respostas nos prazos que mantenham a competitividade das organizações no fornecimento de seus produtos e serviços aos seus clientes.

Mas o que é Governança de TI? Emerson Dorow (2009) explica que é um conjunto de práticas, padrões e relacionamentos estruturados, assumidos por executivos, gestores, técnicos e usuários de TIC de uma organização, com a finalidade de garantir controles efetivos, ampliar os processos de segurança, minimizar os riscos, ampliar o desempenho, otimizar a aplicação de recursos, reduzir os custos, suportar as melhores decisões e consequentemente alinhar TI aos negócios. Serve para auxiliar o Governante de TI, ou seja, o profissional de Tecnologia da Informação (CIO - *Chief Information Officer*) a avaliar os rumos a serem tomados para alcance dos objetivos de sua organização.

Trata-se da área que auxilia o CIO no planejamento, implantação, controle e monitoramento de programas e projetos de governança sob os aspectos operacionais e suas implicações legais.

Por fim, o grande desafio do CIO (*Chief Information Officer*) é o de tornar os processos da organização de forma sincronizada a ponto de demonstrar que a TI não é apenas uma área de suporte ao negócio e sim parte fundamental da estratégia das organizações. Para se ter resultados significativos e que valham o investimento, é necessário avaliar o processo de Governança e outros processos que lhes dão suporte.

Inicialmente, a Governança de TI teve sua maturidade atingida em meados da década de 90 de forma similar à engenharia de *software*. Tem como função solucionar o problema de conflito entre proprietário e gestor da empresa, que em muitos casos seus interesses não estão alinhados.

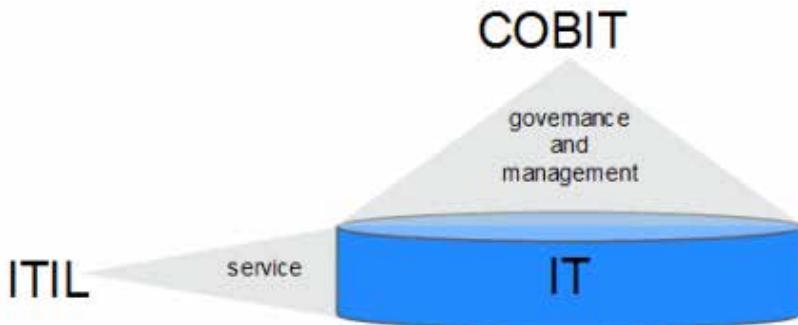
Um dos grandes desafios da Governança de TI, além de manter o alinhamento aos propósitos da organização, é manter também sincronizados os elementos componentes da própria TI, cuja administração deve prezar pela melhor forma de utilizá-los e também estabelecer a garantia da continuidade dos negócios da empresa.

A TIC deve prover através de sua estrutura, meios que estabeleçam a garantia dos processos empresariais mínimos para que a realização dos negócios não sejam comprometidos ou interrompidos, principalmente no momento em que informações essenciais se fizerem necessárias para a efetivação de transações pertinentes à relação direta com seus clientes.

Dois modelos muito bem conhecidos pela área de Governança de TI são o **COBIT** (*Control Objectives for Information and Related Technology*) e o **ITIL** (*Information Technology Infrastructure Library*), sendo o **ITIL** responsável por fornecer as diretrizes para implementação de uma infraestrutura otimizada de TI. Enquanto o **COBIT** é uma documentação para a gestão de TI suprido pelo ISACF (*Informations Systems Audit and Control Foundation*) que visa fornecer informações para gerenciar os processos baseados em seus negócios.

A figura a seguir, mostra a comparação entre os dois modelos ligados à Tecnologia da Informação.

FIGURA 70 – COMPARAÇÃO ENTRE ITIL E COBIT



FONTE: Disponível em: <<http://www.itskeptic.org/files/itskeptic/imagepicker/1/perspective.png>>. Acesso em: 30 set. 2015.

A seguir serão apresentados os dois modelos, COBIT e ITIL.

2 CONTROL OBJECTIVES FOR INFORMATION AND RELATED TECHNOLOGY (COBIT)

O COBIT (*Control Objectives for Information and Related Technology*) é um framework voltado à governança de TI, sua principal função é que a empresa tenha uma visão de forma superficial da área de tecnologia de informação. Sua estrutura se baseia em indicadores de *performance*, podendo monitorar quanto a Tecnologia da Informação está agregando valores aos negócios da organização.

O COBIT define a governança de TI como uma estrutura de relacionamentos entre processos para direcionar e controlar uma empresa de modo a atingir os objetivos corporativos. Para esse gerenciamento e controle o COBIT propõe métodos que utilizam 37 objetivos de controle de alto nível, sendo que, para cada controle são definidos vários objetivos de controle detalhados (VERAS, 2012).

A metodologia COBIT consiste em objetivos de negócio ligados a objetivos de TI, provendo métricas e modelos de maturidade para medir sua eficácia e identificando as responsabilidades relacionadas dos donos dos processos de negócios e de TI (MINGAY, 2015).

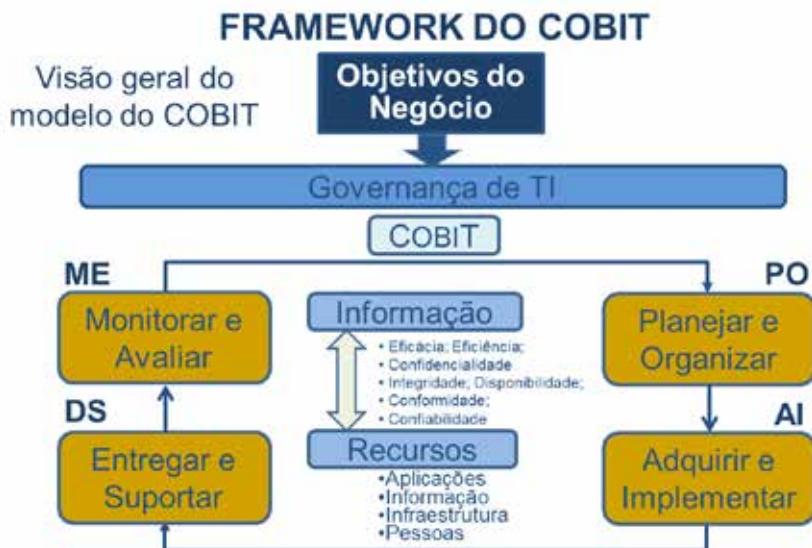
De acordo com o ITGI – *IT Governance Institute* (2015), o principal objetivo das práticas do CobiT é contribuir para o sucesso da entrega de produtos e serviços de TI, a partir da perspectiva das necessidades do negócio com um foco mais acentuado no controle do que na execução. Diz-se que este modelo fornece bases sólidas para o melhor retorno dos investimentos em TI. Então o CobiT:

- Estabelece relacionamentos com os requisitos do negócio.
- Organiza as atividades de TI em um modelo de processos genérico.
- Identifica os principais recursos de TI, nos quais deve haver mais investimento.
- Define os objetivos de controle que devem ser considerados para a gestão.

Atualmente, o COBIT tem sido utilizado por diversas organizações que têm comprometimento e responsabilidade com seus processos de negócios que dependem radicalmente da TI. O modelo foi iniciado e estruturado em 34 processos que estão inter-relacionados, atendendo quatro domínios (com dois objetivos de controle para cada domínio) e estão divididos em: planejar e organizar, adquirir e implementar, entregar e dar suporte, monitorar e avaliar.

A figura a seguir mostra que o modelo COBIT possui quatro domínios: Planejamento e Organização (PO); Aquisição e Implementação (AI); Estrega e Suporte (DS); e Monitoração e Avaliação (ME).

FIGURA 71 – FRAMEWORK DO COBIT



FONTE: Framework do COBIT. Disponível em: <http://www.teclogica.com.br/blog/wp-content/uploads/2012/10/governan%C3%A7a_tI_framework_cobit.png>. Acesso em: 29 set. 2015.

O COBIT tem por missão explícita pesquisar, desenvolver, publicar e promover um conjunto atualizado de padrões internacionais de boas práticas referentes ao uso corporativo da TI para os gerentes e auditores de tecnologia. A metodologia COBIT foi criada pelo ISACA – *Information Systems Audit and Control Association* – através do *IT Governance Institute*, organização independente que desenvolveu a metodologia considerada a base da governança tecnológica.

O COBIT funciona como uma entidade de padronização e estabelece métodos documentados para nortear a área de tecnologia das empresas, incluindo qualidade de *software*, níveis de maturidade e segurança da informação.

Os documentos do COBIT definem Governança Tecnológica como sendo “uma estrutura de relacionamentos entre processos para direcionar e controlar uma empresa de modo a atingir objetivos corporativos, através da agregação de valor e risco controlado pelo uso da tecnologia da informação e de seus processos” (ISACA, 2000).

A Governança Tecnológica considera a área de TI não apenas como um suporte à organização, mas um ponto fundamental para que seja mantida a gestão administrativa e estratégica da organização. O objetivo central é manter processos e práticas relacionados à infraestrutura de sistemas, redes e dispositivos utilizados pela empresa. A análise destes processos deve orientar a organização na decisão de novos projetos e como utilizar tecnologia da informação neles, considerando também a evolução tecnológica, sistemas já existentes, integração com fornecedores, atendimento ao cliente (externo e interno), custo da tecnologia e retorno esperado.

A necessidade de integração de sistemas e a evolução tecnológica são fundamentadas nos processos da metodologia, criando-se métricas para auditoria e medição da evolução das atividades destes processos.

O COBIT está organizado em quatro domínios que podem ser caracterizados pelos seus processos e pelas atividades executadas em cada fase de implantação da Governança Tecnológica. Os domínios do COBIT são: (1) Planejamento e Organização, (2) Aquisição e Implementação, (3) PDI (Plano Diretor de Informática) e (4) Entrega e Suporte.

3 INFORMATION TECHNOLOGY INFRASTRUCTURE LIBRARY (ITIL)

De acordo com Magalhães e Pinheiro (2007, p. 35), “a ITIL é um conjunto de melhores práticas que vem ao encontro do novo estilo de vida imposto às áreas de TI, habilitando o incremento da maturidade do processo de gerenciamento de TI”. A ITIL é nada mais que um guia de boas práticas desenvolvido para auxiliar as organizações que pretendem desenvolver melhorias em seus processos, pois a ITIL fornece orientação para todos os tipos de provedores de serviço de TI. No entanto, não é obrigatório implantar tudo que está escrito nele, mas sim, deve ser utilizado como um guia de apoio e melhoria de acordo com as necessidades de cada organização.

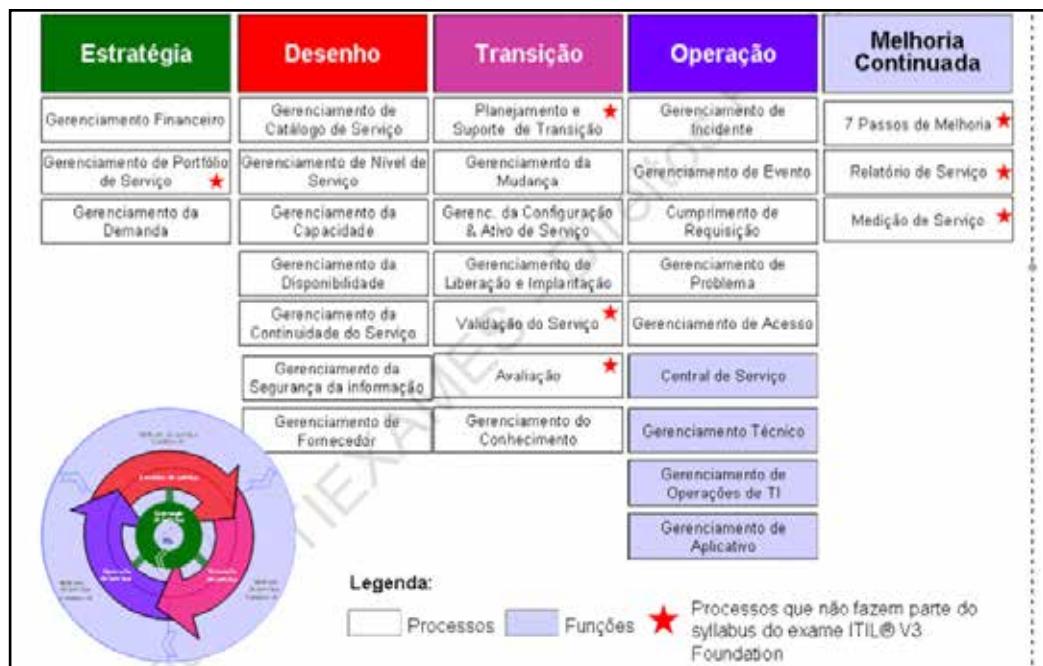
Cabe ressaltar que a ITIL vem se destacando como padrão no gerenciamento de serviços de TI pelas organizações, pois a mesma oferece recomendações de como planejar, gerenciar, controlar e melhorar os serviços de TI necessários para a diferenciação da empresa em relação ao mercado externo.

A biblioteca ITIL foi criada na década de 90, com base na qualidade dos serviços de TI oferecidos pelo governo britânico, então foi solicitado à antiga CCTA (*Central Computer and Telecommunications Agency*), atualmente OGC, que disponibilizasse procedimentos para o setor público inglês aperfeiçoar o uso dos recursos de TI, garantirem melhores resultados e ser competente nos custos.

Apesar de cada modelo ter um foco diferente, eles não são mutuamente excludentes e podem ser combinados para prover um melhor gerenciamento da tecnologia, garantindo não só o suporte tecnológico necessário para que a organização atinja seus objetivos estratégicos com qualidade e preço competitivo, mas também a satisfação dos seus clientes (MINGAY; BITTINGER, 2002).

A figura a seguir mostra os ciclos de vida e objetivos do ITIL v3, apresentando os processos e funções que compõem todo o processo.

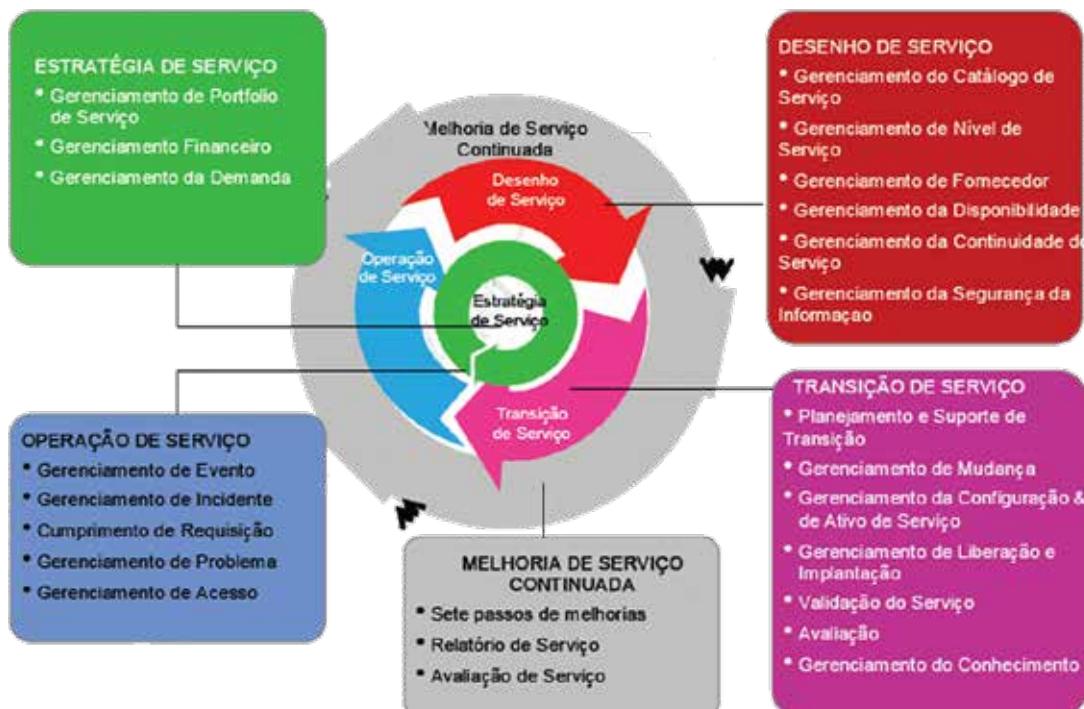
FIGURA 72 – PROCESSOS E FUNÇÕES DO ITIL V3



FONTE: Disponível em: <http://api.ning.com/files/3GrJHH7qBsTS6HfSqbpa1iOXZtTdeE-ojXxOB8dFP2DnrWluQE1NzgJNppM5vD8OigcFz5T6P9x3UT-uNtll6Bbj11WHLA/itil_v3.png>. Acesso em: 30 set. 2015.

A ITIL v3 é composta por cinco publicações do ciclo de vida do serviço.

FIGURA 73 – CICLO DE VIDA DO SERVIÇO DE ITIL V3



FONTE: Disponível: <<http://www.pedrofcarvalho.com.br/itiservicos.png>>. Acesso em: 30 set. 2015.

Segundo Pinheiro (2011), as principais organizações envolvidas com a ITIL® são:

- *Cabinet Office - Proprietária atual da ITIL®.*
- *TSO (The Stationery Office) – É a editora que publica as obras da Coroa Britânica.*
- *ItSMF (it Service Management Forum) – Organização sem fins lucrativos que realiza as publicações da ITIL pelo mundo todo.*
- *APMG - É a credenciada oficial da ITIL, ela é a responsável por realizar a distribuição da ITIL para as instituições, gerencia todo o esquema de certificação e a marca ITIL®.*

Por último existem as instituições que oferecem o treinamento da ITIL para profissionais interessados na área.

Segundo Freitas (2010), a ITIL se iniciou na década de 80, no Reino Unido, a CCTA (*Central Computer and Telecommunications Agency*) ou Agência Central de Computadores e Telecomunicações desenvolveu o GITM (*Government Information Technology Infrastructure Method*) ou Método de Governo de Infraestrutura de Tecnologia da Informação. O objetivo do GITM era atender à crescente dependência do governo em TI no que diz respeito à padronização de práticas de TI.

O governo do Reino Unido percebeu que havia um risco de que o mercado, e, principalmente, as empresas privadas que mantinham contratos com o governo, utilizassem suas práticas próprias de gestão de TI e isso poderia gerar esforços duplicados e desentendimentos que resultariam em custos excessivos de TI.

Em 1989, o GITM foi renomeado para ITIL® por causa da palavra “método” no nome GITM. Essa iniciativa foi resultado do crescente interesse de outras entidades externas ao governo britânico nas práticas propostas e tinha como objetivo refletir a visão de que o ITIL® não era uma regra, mas sim uma recomendação ou guia.

Ao longo dos anos, melhorias foram implantadas ao ITIL e consequentemente versões foram surgindo e se difundindo em vários módulos de acordo com cada necessidade.

O ITIL versão 1 ou ITIL v1 era composto por 31 livros que abordavam aspectos fundamentais para a provisão dos serviços de TI e foi utilizado principalmente no Reino Unido e na Holanda. Sua abrangência ia do cabeamento de redes ao planejamento de contingência de sistemas de TI (FREITAS, 2010).

A partir do ano 2000, o ITIL® foi revisado e foi publicado na versão 2. O ITIL v2 consistia em sete livros que cobriam os aspectos relativos aos processos de gerenciamento de serviços de TI com uma forte orientação em processos e melhoria contínua (ciclo PDCA – *Plan, Do, Check, Act*).

O conteúdo dos 31 livros foi organizado em uma função e 10 processos relacionados entre si. O v2 foi difundido e aceito mundialmente, sendo reconhecido como padrão universal de gerenciamento de serviços de TI. Entre 2007 e 2008 foi lançada a versão 3 do ITIL. O ITIL v3 é composto de cinco livros onde a visão de processos da v2 foi organizada em ciclos de vida contendo cinco fases. (FREITAS, 2010).

O ITIL é uma marca registrada do *The Cabinet Office* no Reino Unido e em outros países. Gerenciamento de Serviços de TI com base em ITIL V3.

Em ITIL v3, um serviço é um meio de entregar valor ao cliente, facilitando os resultados que o cliente deseja alcançar, sem ter que assumir custos e riscos. E serviço de TI se caracteriza por ser fornecido por um provedor de serviços de TI, composto por uma combinação tecnológica, (PINHEIRO, 2011).

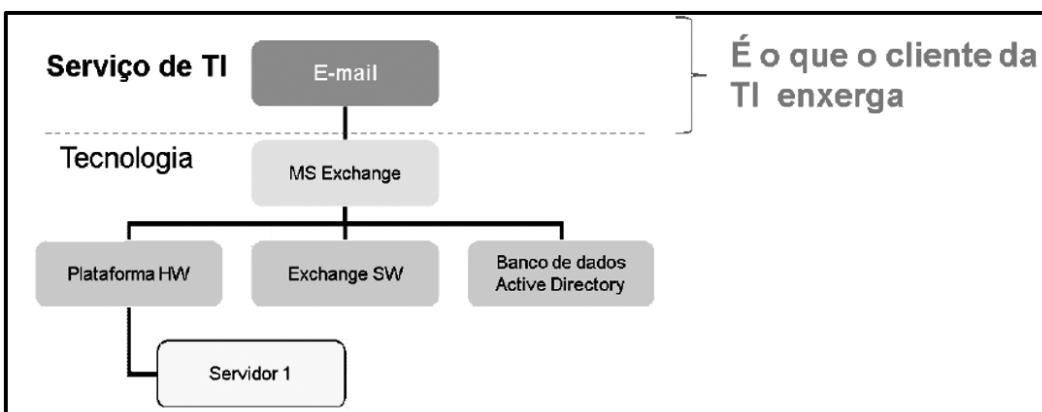
FIGURA 74 – REPRESENTAÇÃO DE COMBINAÇÃO TECNOLÓGICA PROPOSTA PELA ITIL



FONTE: Adaptado de: Pinheiro (2011)

O usuário final não sabe o que é necessário para fazer o serviço funcionar, mas enxerga o resultado final. Utiliza-se a figura a seguir para representar um exemplo de serviço de *e-mail* disponibilizado para uma organização.

FIGURA 75 – VISÃO TI VERSUS USUÁRIO



FONTE: Adaptado de: Pinheiro (2011)

Uma organização possui inúmeros tipos de serviços de TI (*ERP - Enterprise Resource Planning*, *E-mail*, Internet, Manutenções, entre outros), e cabe somente a TI oferecer e manter qualidade na entrega desses serviços.

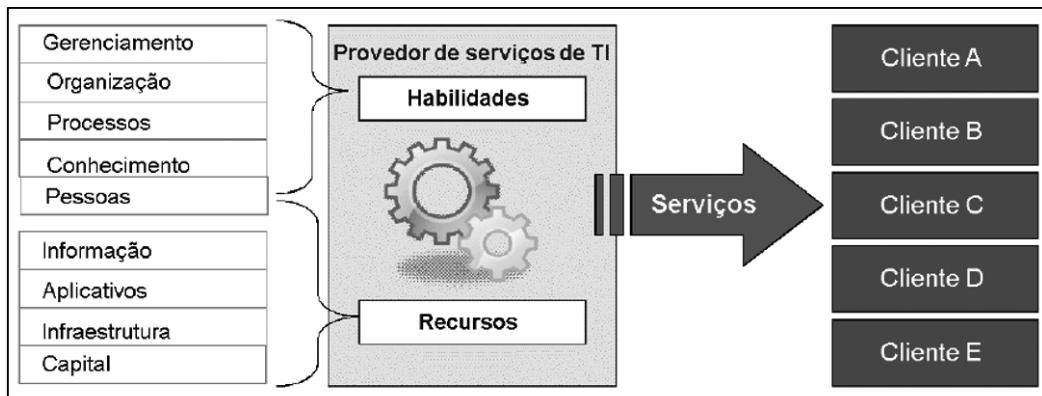
Deste modo, no meio de tantos serviços, a TI precisa encontrar um modo de gerenciá-los. A ITIL v3 define o gerenciamento de serviços como sendo um conjunto de habilidades organizacionais para fornecer valor aos clientes em forma de serviços. Garantindo ao provedor de serviços entender os serviços que estão sendo fornecidos, facilitando o alcance dos resultados que seus clientes querem alcançar, visando os custos e riscos (PINHEIRO, 2011). O gerenciamento de serviços é nada mais que uma gestão interna para a entrega de um serviço, buscando a solução por meios internos ou não.

Trazendo este cenário para a TI, Pinheiro (2011) diz que a ITIL v3 define o termo de gerenciamento de serviços de TI, como sendo a implementação e o gerenciamento da qualidade dos serviços de TI de forma a atender às necessidades do negócio, por meio de combinações de pessoas, processos e tecnologia da informação.

Essas combinações são as habilidades que se encontram para transformar recursos em serviços de valor, modificando-as conforme a necessidade do negócio fornecida pelo cliente. O termo “Negócio” é nada mais que uma organização para qual o provedor de serviços de TI fornece seus serviços.

A figura a seguir exemplifica o esquema do gerenciamento de TI proposta pela ITIL v3.

FIGURA 76 – ILUSTRAÇÃO DE ESQUEMA DO GERENCIAMENTO DE SERVIÇO DE TI



FONTE: Adaptado de: Pinheiro (2011)

3.1 PROVEDOR DE SERVIÇOS DE TI

Um provedor de serviços de TI fornece serviços para clientes internos ou externos. A ITIL v3 descreve três tipos de provedores (PINHEIRO, 2011):

- **Provedores de serviços internos** – está localizado dentro de cada unidade de negócio, podendo ser várias dentro da organização. Isto se refere às empresas com filiais, neste modo cada filial terá um núcleo de TI dentro de cada unidade.

- **Provedores de serviços compartilhados** – fornece os serviços de TI para várias unidades de negócio.
- **Provedores de serviços externos** – fornece serviços de TI para clientes externos.

Cabe adiantar que a orientação da ITIL v3 não impede que uma organização possa possuir vários tipos de provedores de serviços de TI dentro dela mesma, pois cada tipo de serviço pode necessitar de recursos diferentes que nem a própria organização pode oferecer internamente (PINHEIRO, 2011).

Os *stakeholders* são os interessados em adquirir estes serviços, eles podem ser internos ou externos, a ITIL v3 também realiza distinções entre eles (PINHEIRO, 2011):

- **Internos** – São aqueles que trabalham para o mesmo negócio que o provedor de serviços de TI. É composto pelos diversos setores e colaboradores dentro da organização.
- **Externos** – São clientes que trabalham para um negócio diferente do provedor de serviços de TI. São os nossos fornecedores, aqueles que oferecem serviços para dentro de nossa organização.

3.2 TIPOS DE SERVIÇOS DE TI

Um serviço possui três níveis de relacionamento entre si ou com seus usuários (PINHEIRO, 2011):

- **Serviços principais** – São os serviços entregues para o cliente.
- **Serviços de Apoio** – São os serviços que estão por trás, para fazer o serviço principal funcionar, como por exemplo, a internet para o funcionamento do *e-mail*.
- **Serviços Intensificadores** – Usados para criar diferenciação entre produtos. Isso se refere à criação de funcionalidades extras, não sendo necessárias para o funcionamento do serviço principal, mas que se torna atraente para o cliente.

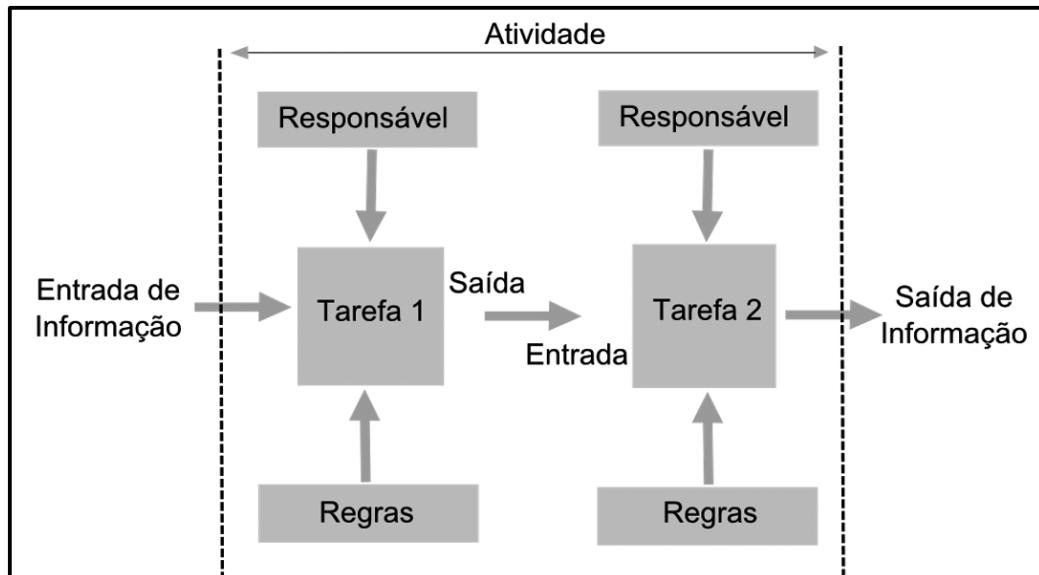
3.3 PROCESSOS E FUNÇÕES

No gerenciamento de serviços de TI existem várias atividades, a ITIL v3 agrupa essas atividades em processos. Um processo é formado por diversas atividades que integram para o alcance do objetivo especificado e a geração do resultado desejado.

Para a ITIL v3 os processos são o mais alto nível de definição de atividades de uma organização. Os procedimentos são mais detalhados e descrevem exatamente o que deve ser executado em determinada atividade do processo (MAGALHÃES; PINHEIRO, 2007).

Dentro de um processo pode haver vários departamentos da organização. A ITIL v3 define esses departamentos ou pessoas como funções. Os processos facilitam a comunicação entre as funções para resolver determinadas atividades. Cada processo envolve uma entrada e saída e, a partir da entrada da informação, as atividades são desenvolvidas gerando a consequente saída, (MAGALHÃES; PINHEIRO, 2007). O modelo ilustrado da figura a seguir, mostra como funciona uma atividade que se compõe dentro de um processo.

FIGURA 77 – COMPOSIÇÃO DE UMA ATIVIDADE



FONTE: Modelo ilustrativo Magalhães e Pinheiro – Gerenciamento de Serviços de TI na prática (2007)

3.4 PAPÉIS

Em gerenciamento de serviços, um papel é um conjunto de responsabilidades, atividades e autorizações concedidas a uma pessoa ou equipe. Uma pessoa pode possuir vários papéis. Deste modo, pessoas assumem papéis em processos ou serviços (PINHEIRO, 2011).

Portanto, a ITIL v3 descreve papéis genéricos no ciclo de vida do serviço.

- **Dono do processo:** pode ser o próprio cliente ou não, ele é o responsável por verificar se o processo está apto ao seu propósito, define as estratégias, patrocina o projeto etc. Em geral é quem solicitou o projeto ou é o total responsável a assumir os riscos do processo. Cada processo deve haver apenas um dono (PINHEIRO, 2011).
- **Gerente do processo:** é responsável pelo gerenciamento operacional. Assegura que as atividades estão sendo desenvolvidas corretamente. Para cada processo pode haver vários gerentes (PINHEIRO, 2011).

- **Profissional do processo:** é o responsável pela realização das atividades do processo (PINHEIRO, 2011).
- **Dono do serviço:** assegura que o serviço é gerenciado com o foco no negócio, e também é responsável pela entrega deste serviço. O mesmo presta contas com o dono do serviço, está por dentro do negócio, entre outros (PINHEIRO, 2011).

Para cada processo, os papéis são distribuídos conforme as responsabilidades de cada pessoa, podendo ou não ser atribuídos mais papéis por pessoa.

3.5 MATRIZ RACI (*RESPONSIBLE, ACCOUNTABLE, CONSULTED, INFORMED*)

A ITIL indica uma ferramenta para ajuda no controle de qualidade na execução de cada projeto, conhecida em português como matriz RPCI (Responsável, Prestador de contas, Consultado, Informado) ou em inglês como RACI. A matriz RPCI estabelece quatro atribuições em relação à processos e atividades (SILVA; GOMEZ; MIRANDA, 2010).

- **R - Responsável:** é o responsável por executar a atividade do processo.
- **P - Prestador de contas:** é aquele que assume as responsabilidades pelo resultado final do processo.
- **C - Consultado:** é a pessoa que fornece informações ao longo do ciclo de vida do processo.
- **I - Informado:** é a pessoa que fica atualizada de todo o andamento do processo.

A matriz ajuda a definir os papéis e responsabilidades dos processos.

LEITURA COMPLEMENTAR

Um dos problemas mais críticos, latentes e desafiadores para os gestores das organizações são, sem dúvida, conseguir responder à pergunta: como alinhar negócios e TI?

Como leitura complementar, a seguir, destaca-se um artigo que aborda importantes considerações sobre o tema “Como alinhar negócios e TI?”, de autoria de Wilson Caldeira da Silva*. O autor faz algumas reflexões interessantes que valem a pena ler.

Como alinhar negócios e TI?

Diariamente várias questões povoam os pensamentos de responsáveis pela área de Tecnologia da Informação de empresas e, certamente, algumas dessas dúvidas incomodam mais do que outras. Dentre as que mais contribuem para as “dores de cabeça gerenciais”, pode-se destacar duas:

- De que forma a área de TI pode ser vista como suporte imprescindível para a geração de novas receitas para a empresa?
- Existe alguma linha tecnológica que possa ser colocada em prática para facilitar o alinhamento entre modelos de gestão de negócios e de tecnologia?

É tão difícil encontrar as respostas para essas perguntas, quanto responder a questionamentos sobre nossa existência (Quem somos? De onde viemos? Para onde vamos?). Apesar disso, a exemplo das dúvidas sobre a existência humana, o questionamento da existência da empresa como organismo vivo pode e deve ser continuamente exercitado e avaliado.

Centro de Receitas

Nessa reflexão, pode-se perceber que tradicionalmente a área de TI das empresas é vista como centro de custos, onde uma quantidade não desprezível de recursos flui continuamente. E isto acontece principalmente quando geração de novas receitas, cortes direto de custos ou aumento de produtividade não são diretamente relacionados aos investimentos já feitos em TI.

Organizações mais tradicionais ainda relutam em inserir em suas discussões sobre posicionamento estratégico as oportunidades e os limites gerados por sua estrutura de TI. Assim, torna-se quase impossível perceber oportunidades de negócio geradas com suporte direto da área de Tecnologia da Informação, ou mesmo antecipar problemas em novos negócios que podem ser criados pelas limitações tecnológicas da estrutura existente. Temos aí um desafio: como tornar estratégico algo que é visto como periférico?

Mudança de Paradigma

Estruturas empresariais mais modernas (podemos citar os setores de telecomunicação e bancos como exemplos) já concluíram que não podem separar seus produtos (ou serviços) da tecnologia. Basta olharmos com atenção para todos os produtos lançados por estes segmentos para comprovarmos esta afirmação: não existe mais produto bancário ou da área de telecomunicações sem o suporte direto da área de Tecnologia da Informação.

Além disso, ao analisarmos a estrutura de poder dentro destes setores no país, veremos que a posição ocupada pelas altas decisões de investimentos em TI é estratégica e diretamente relacionada a tomadas de decisão em diversas linhas de negócio.

E por que outros grandes segmentos do mercado ainda não agem da mesma forma? Talvez porque a dinâmica de negócios em alguns setores seja evidentemente mais lenta do que em outros, talvez porque o próprio posicionamento da área de TI nestes setores mais conservadores, ainda não encontrou uma forma para ocupar o espaço que lhe é devido.

Sejam quais forem os motivos deste desalinhamento estratégico, a área de TI pode e deve tomar algumas medidas para fortalecer sua posição de comprometimento com os objetivos do negócio. Confira a seguir que medidas são essas.

Primeiramente, podemos citar a necessidade da área de TI analisar, de forma contínua e estratégica, a contribuição que os produtos gerados por sua equipe (sistemas de informação) têm em relação aos objetivos do negócio. Esta análise deve ser feita em relação ao orçamento total da área de TI. Devem ser quantificados e analisados os percentuais investidos em três blocos funcionais distintos:

- Processos que não agregam valor ao negócio, por serem obrigatórios em todas as empresas de um mesmo segmento (como contas a pagar, contabilidade etc.).
- Processos que aumentam a produtividade da empresa (normalmente sistemas ligados à linha de produção ou atendimento ao cliente).
- Processos que efetivamente contribuem para a geração de novos negócios para a empresa.

Este tipo de análise servirá como um painel para a área de TI enxergar e fazer com que a empresa entenda que sua participação em relação aos objetivos do negócio pode ser expandida.

“Negocês” x “tecniquês”

Outra medida que pode contribuir fortemente para este reposicionamento é a adoção de linhas tecnológicas que simplifiquem a tradução do modelo de negócios em ferramentas de TI, já que o dialeto “negocês” (retorno de investimento, margem bruta, canais de distribuição etc.) terá que ser implementado com o dialeto “tecniquês” (*Java, Web Server, Banco de Dados etc.*).

Além disso, sempre é bom lembrarmos que existe no mercado um leque de soluções tecnológicas que, inicialmente, fazem a alegria do pessoal de TI, pois incorporam todas as promessas de ser “a ferramenta para acabar com todas as outras ferramentas”. No entanto, no momento de entrar em ação, de atender à área de negócios com agilidade e cursos baixos, essas “ferramentas mágicas” mostram dificuldades de operação por profissionais que não sejam extremamente qualificados.

Caso esta tradução de modelos seja complexa, a área de TI se encontrará várias vezes na situação em que uma pequena mudança no modelo de negócios da empresa acarretará uma grande mudança no modelo tecnológico. Este tipo de situação é visto pela área de negócios como um investimento sem retorno, já que pequenas mudanças no negócio devem ser executadas em prazo curto e com orçamento pequeno.

Foco no negócio

A terceirização das atividades de TI que não possuam valor agregado aos objetivos de negócio da empresa é o outro ponto-chave nesta questão. Será que uma empresa, pelo simples fato de utilizar uma salada tecnológica necessária para a implementação de seu modelo de negócios, precisa ter dentro de casa todos os profissionais especializados em cada uma dessas tecnologias? Provavelmente alguma outra empresa focada no ramo de TI terá condições de executar estas atividades com uma relação custo/benefício bastante atraente.

Em resumo, pode-se dizer, mais do que nunca, que o mundo dos negócios precisa entender o papel que TI pode desempenhar; que a Tecnologia da Informação precisa entender os objetivos do negócio, estratégias e planos; e que os dois lados precisam trabalhar juntos o tempo todo, desde o começo do ciclo de planejamento dos negócios, até a implementação final da solução de TI.

*Gerente de Marketing da MSA-INFOR e diretor Comercial da Datran, empresas MSA. Atua no mercado de TI há mais de 23 anos e também tem experiência na área técnica. É bacharel em Matemática, com especialização em Informática pela UFRJ, e é pós-graduado em Marketing, pela FGV.

FONTE: Disponível em: <http://www.timaster.com.br/revista/artigos/main_artigo.asp?codigo=497&pag=2>. Acesso em: 30 out. 2015.

RESUMO DO TÓPICO 4

Neste tópico, você aprendeu que:

- Governança de TI é um conjunto de práticas, padrões e relacionamentos estruturados, assumidos por executivos, gestores, técnicos e usuários de TIC para garantir controles efetivos, suportar as melhores decisões e consequentemente alinhar TI aos negócios.
- Trata-se da área que auxilia o CIO no planejamento, implantação, controle e monitoramento de programas e projetos de governança sob os aspectos operacionais e suas implicações legais.
- O COBIT e o ITIL são os dois modelos mais recomendados para a área de Governança de TI:
 - ✓ o COBIT é uma documentação para a gestão de TI suprido pelo ISACF (*Informations Systems Audit and Control Foundation*) que visa fornecer informações para gerenciar os processos baseados em seus negócios.
 - ✓ o ITIL é responsável por fornecer as diretrizes para implementação de uma infraestrutura otimizada de TI.
- O COBIT (*Control Objectives for Information and Related Technology*) é um framework voltado à governança de TI, sua principal função é que a empresa tenha uma visão de forma superficial da área de tecnologia de informação. Sua estrutura se baseia em indicadores de *performance*, podendo se monitorar o quanto a Tecnologia da Informação está agregando valores aos negócios da organização.
- O principal objetivo das práticas do CobiT é contribuir para o sucesso da entrega de produtos e serviços de TI, a partir da perspectiva das necessidades do negócio com um foco mais acentuado no controle do que na execução. Diz-se que este modelo fornece bases sólidas para o melhor retorno dos investimentos em TI.
- O COBIT funciona como uma entidade de padronização e estabelece métodos documentados para nortear a área de tecnologia das empresas, incluindo qualidade de *software*, níveis de maturidade e segurança da informação.
- O COBIT está organizado em quatro domínios que podem ser caracterizados pelos seus processos e pelas atividades executadas em cada fase de implantação da Governança Tecnológica. Os domínios do COBIT são: (1) Planejamento e Organização, (2) Aquisição e Implementação, (3) PDI (Plano Diretor de Informática) e (4) Entrega e Suporte.

- A ITIL é um conjunto de melhores práticas que vem ao encontro do novo estilo de vida imposto às áreas de TI, habilitando o incremento da maturidade do processo de gerenciamento de TI. A ITIL é nada mais que um guia de boas práticas desenvolvido para auxiliar as organizações que pretendem desenvolver melhorias em seus processos, pois a ITIL fornece orientação para todos os tipos de provedores de serviço de TI.
- A biblioteca ITIL foi criada na década na 90, com base na qualidade dos serviços de TI oferecidos pelo governo britânico, então foi solicitado à antiga CCTA (*Central Computer and Telecommunications Agency*), atualmente OGC, que disponibilizasse procedimentos para o setor público inglês aperfeiçoar o uso dos recursos de TI, garantirem melhores resultados e ser competente nos custos.
- Na última versão da ITIL, a v3, um serviço é um meio de entregar valor ao cliente, facilitando os resultados que o cliente deseja alcançar, sem ter que assumir custos e riscos. E serviço de TI se caracteriza por ser fornecido por um provedor de serviços de TI, composto por uma combinação tecnológica.
- A ITIL v3 define o gerenciamento de serviços como sendo um conjunto de habilidades organizacionais para fornecer valor aos clientes em forma de serviços. Garantindo ao provedor de serviços entender os serviços que estão sendo fornecidos, facilitando o alcance dos resultados que seus clientes querem alcançar, visando aos custos e riscos. O gerenciamento de serviços é nada mais que uma gestão interna para a entrega de um serviço, buscando a solução por meios internos ou não.
- Um provedor de serviços de TI fornece serviços para clientes internos ou externos:
 - ✓ Provedores de serviços internos – estão localizados dentro de cada unidade de negócio, podendo ser vários dentro da organização. Isto se refere às empresas com filiais, neste modo cada filial terá um núcleo de TI dentro de cada unidade.
 - ✓ Provedores de serviços compartilhados – fornece os serviços de TI para várias unidades de negócio.
 - ✓ Provedores de serviços externos - fornece serviços de TI para clientes externos.
- Um serviço possui três níveis de relacionamento entre si ou com seus usuários:
 - ✓ Serviços principais – são compostos pelo serviço entregue para o cliente.
 - ✓ Serviços de Apoio – são os serviços que estão por trás, para fazer o serviço principal funcionar, como por exemplo, a internet para o funcionamento do e-mail.
 - ✓ Serviços Intensificadores – usados para criar diferenciação entre produtos. Isso se refere à criação de funcionalidades extras, não sendo necessárias para o funcionamento do serviço principal, mas que se torna atraente para o cliente.

- Os processos e funções no ITIL facilitam a comunicação entre as funções para resolver determinadas atividades. Cada processo envolve uma entrada e saída e, a partir da entrada da informação, as atividades são desenvolvidas gerando a consequente saída.
- Um papel no ITIL é um conjunto de responsabilidades, atividades e autorizações concedidas a uma pessoa ou equipe. Uma pessoa pode possuir vários papéis. Deste modo, pessoas assumem papéis em processos ou serviços.
- A ITIL v3 descreve papéis genéricos no ciclo de vida do serviço:
 - ✓ Dono do processo: pode ser o próprio cliente ou não, ele é o responsável por verificar se o processo está apto ao seu propósito, define as estratégias, patrocina o projeto etc. Em geral é quem solicitou o projeto ou é o total responsável a assumir os riscos do processo. Cada processo deve haver apenas um dono.
 - ✓ Gerente do processo: é responsável pelo gerenciamento operacional. Assegura que as atividades estão sendo desenvolvidas corretamente. Para cada processo pode haver vários gerentes.
 - ✓ Profissional do processo: é o responsável pela realização das atividades do processo.
 - ✓ Dono do serviço: assegura que o serviço é gerenciado com o foco no negócio, e também é responsável pela entrega deste serviço. O mesmo presta contas com o dono do serviço, está por dentro do negócio, entre outros.
- A ITIL indica uma ferramenta para ajuda no controle de qualidade na execução de cada projeto, conhecida em português como matriz RPCI (Responsável, Prestador de contas, Consultado, Informado) ou em inglês como RACI. A matriz RPCI estabelece quatro atribuições em relação a processos e atividades:
 - ✓ R - Responsável: é o responsável por executar a atividade do processo.
 - ✓ P - Prestador de contas: é aquele que assume as responsabilidades pelo resultado final do processo.
 - ✓ C - Consultado: é a pessoa que fornece informações ao longo do ciclo de vida do processo.
 - ✓ I - Informado: é a pessoa que fica atualizada de todo o andamento do processo.



1 O principal objetivo da Governança de TI é:

- a) () Entender as estratégias do negócio e traduzi-las em planos para sistemas, aplicações, soluções, estrutura e organização, processos e infraestrutura.
- b) () Alinhar TI aos requisitos do negócio. Este alinhamento tem como base a continuidade do negócio, o atendimento às estratégias do negócio e o atendimento a marcos de regulação externos.
- c) () Implantar os projetos e serviços planejados e priorizados.
- d) () Prover a TI da estrutura de processos que possibilite a gestão do seu risco para a continuidade operacional da empresa.

2 Um dos modelos utilizados para governança de TI tem o propósito de sincronizar as necessidades do negócio com a gestão da tecnologia, com o foco na qualidade dos serviços prestados. O modelo descrito é:

- a) () ITIL.
- b) () Cobit.
- c) () PMBOK.
- d) () Scrum.

3 A missão do COBIT é:

- a) () promover o desenvolvimento da padronização e das atividades relacionadas à qualidade com o objetivo de facilitar a troca ou comercialização de produtos e serviços e desenvolver cooperação na esfera intelectual, científica, tecnológica e econômica.
- b) () promover a segurança da informação, seus objetivos gerais, seu escopo e a importância dessa segurança como um mecanismo que possibilite o compartilhamento de informações.
- c) () criar um padrão para gerenciar a maioria dos projetos, na maior parte das vezes, em vários tipos de setores de indústria, e descrever os processos, as ferramentas e técnicas de gerenciamento de projetos usados até a obtenção de resultados bem-sucedidos.
- d) () pesquisar, desenvolver, publicar e promover um modelo de governança de TI atualizado e internacionalmente reconhecido para ser adotado por organizações.

4 No contexto da ITIL, analise os itens a seguir:

- I. Provedor de serviços internos.
- II. Unidade de serviços compartilhados.
- III. Provedor de serviços externos.

São tipos de provedores de serviços descritos pela ITIL?

- a) () Apenas II.
- b) () Apenas I e II.
- c) () Apenas II e III.
- d) () I, II e III.

5 O ciclo Operação de Serviço, do ITIL, é composto por vários processos e funções. Assinale os dois processos do ciclo operação de serviços.

- a) () Central de serviços e gerenciamento de incidentes.
- b) () Gerenciamento técnico e gerenciamento de problemas.
- c) () Gerenciamento de eventos e cumprimento de requisição.
- d) () Gerenciamento de operações de TI e gerenciamento de acesso.

6 A Tabela ou Matriz RACI descreve os papéis e as responsabilidades dos processos identificando, necessariamente,

- a) () responsáveis pela execução das atividades, aprovadores das atividades, consultados sobre informações para a realização das atividades e informados sobre o andamento das atividades.
- b) () responsáveis pela execução das atividades, patrocinadores das atividades, revisores das atividades e informados sobre o andamento das atividades.
- c) () revisores das atividades, aprovadores das atividades, elaboradores das atividades e patrocinadores das atividades.
- d) () revisores da execução das atividades, auditores do processo, consultados sobre informações para a realização das atividades e informados sobre o andamento das atividades.

REFERÊNCIAS

{TR}SAMPAIO. ISO/IEC 15504. 2014. Disponível em: <<http://www.trsampaio.com/2014/08/27/resumo-iso-15504-spice/>>. Acesso em: 23 set. 2015.

AGILE ALLIANCE. Disponível em: <<http://www.agilealliance.org/>>. Acesso em: 29 jun. 2015.

AGILEMANIFESTO. Disponível em: <<http://www.agilemanifesto.org>>. Acesso em: 16 set. 2015.

ALEXANDROU, Marios. **Dynamic Systems Development Model (DSDM) Methodology**. Disponível em: <<http://www.mariosalexandrou.com/methodologies/dynamic-systems-development-model.asp>>. Acesso em: 21 jun. 2015.

AMARAL, Daniel Capaldo. **Gerenciamento Ágil de Projetos: Aplicação em Produtos Inovadores**. São Paulo: Saraiva, 2011.

AMBLER, Scott W. **Modelagem Ágil. Práticas eficazes para programação extrema e o processo unificado**. Porto Alegre: Bookman, 2003.

AMPARO, Evandro da Silva. **Erros e acertos no desenvolvimento de software**. Disponível em: <<http://blog.tecsystem.com.br/index.php/erros-e-acertos-no-desenvolvimento-de-software/>>. Acesso em: 5 jun. 2015.

ANANIAS, Rafael R. T. **Estudo comparativo entre ferramentas de Gerência de Requisitos**. 2009. 66 f. Trabalho de Conclusão de Curso (Curso de Ciência da Computação) - Universidade Federal de Pernambuco, Recife, 2009. Disponível em: <<http://www.cin.ufpe.br/~tg/2009-2/rrta.pdf>>. Acesso em: 15 jun. 2015.

ANTONIONI, José A. **MPS.BR: Melhoria de Processo do Software Brasileiro**. Xerém, 2007. Disponível em: <http://www.inmetro.gov.br/painelsetorial/palestras/melhoria_software.pdf>. Acesso em: 26 set. 2015.

ARAÚJO, Fábio. **Manual de Gestão de Riscos**. Disponível em: <http://pt.slideshare.net/fabiocdaraujo?utm_campaign=profiletracking&utm_medium=sssite&utm_source=sslidetitle>. Acesso em: 27 ago. 2015.

ARLOW, Jim; NEUSTADT, Ila. **Enterprise Patterns and MDA – Building Better Software with Archetype Patterns and UML**. Addison-Wesley, 2003.

ARTEIRO, Iveruska C. J. B. **Melhoria de processo de Engenharia de Requisitos em empresas de mercado através de transferência de tecnologia**. Monografia (Pós-graduação em Ciência da Computação.) - Centro de Informática, Universidade Federal de Pernambuco. Disponível em: <http://www.cin.ufpe.br/~in1020/arquivos/monografias/2013_2/iveruska.pdf>. Acesso em: 24 jul. 2015.

AZEVEDO, Douglas José Peixoto. **Evolução de software**. Disponível em: <<http://www.batebyte.pr.gov.br/modules/conteudo/conteudo.php?conteudo=299>>. Acesso em: 30 jun. 2015.

BARDINE, Renan. **ISO 9000**. Disponível em:<<http://www.coladaweb.com/administracao/iso-9000>>. Acessado em: 23 set. 2015.

BARTIE, Alexandre. **Artigo Engenharia de Software – Agilidade ou Controle Operacional? Os dois! Engenharia de Software Magazine**. Disponível em: <<http://www.devmedia.com.br/artigo-engenharia-de-software-agilidade-ou-controle-operacional-os-dois/8031>>. Acessado em: 23 set. 2015.

BARTIÉ, Alexandre. **Garantia da Qualidade de Software**. Rio de Janeiro. Editora Campus, 2002.

BATEBYTE. **Aderência do RUP à norma NBR ISO/IEC 12207**. Disponível em: <<http://www.batebyte.pr.gov.br/modules/conteudo/conteudo.php?conteudo=325>>. Acessado em: 24 set. 2015.

BERNARDES, Rodrigo Idiarte. **Trabalho de Web II: era da computação**. Disponível em: <<http://www.geocities.ws/trabalhoweb2/>>. Acesso em: 3 jul. 2015.

BUENO, Jonatas. **DSDM: Metodologia de Desenvolvimento de Sistemas Dinâmicos**. 2014. Disponível em: <<https://prezi.com/9oyzbeuh7yaz/dsdm-metodologia-de-desenvolvimento-de-sistemas-dinamicos/>>. Acesso em: 26 set. 2015.

CABRAL, Kevin. Conceitos Básicos: **Gerência de configuração**. Disponível em: <<http://slideplayer.com.br/slide/3663751/>>. Acesso em: 26 ago. 2015.

CARVALHO, Ariadne M. B. Rizzoni; CHIOSSI, Thelma C. dos Santos. **Introdução à engenharia de software**. CAMPINAS: UNICAMP, 2001. 148p.

CARVALHO, M. M.; RABECHINI Jr, R. **Fundamentos em gestão de projetos: construindo competências para gerenciar projetos**. 3. ed. São Paulo: Editora Atlas, 2011.

CRAIG, R.D., JASKIEL, S. P. **Systematic Software Testing**. Artech House Publishers, Boston, 2002.

DA SILVA FILHO, Antônio Mendes. **Gestão de Projetos de Software**. Disponível em: <<http://www.devmedia.com.br/gestao-de-projetos-de-software/9143>>. Acesso em: 23 jun. 2015.

DANTAS, Cristine. **Gerência de Configuração de Software**. Disponível em: <<http://www.devmedia.com.br/gerenciadeconfiguracaodesoftware/9145>>. Acesso em: 2 jul. 2015.

DIAS NETO, Arilo Cláudio. **Engenharia de Software Magazine – Introdução a Teste de Software**. Disponível em: <<http://www.comp.ita.br/~mluisa/TesteSw.pdf>>. Acesso em: 20 fev. 2015.

DIAS, André Felipe. **O que é Gerência de Configuração?** Disponível em: <http://www.pronus.eng.br/artigos_tutoriais/gerencia_configuracao/gerencia_configuracao.php>. Acesso em: 30 jul. 2015.

DIJKSTRA, E. W. The Humble Programmer: Turing Award Lecture, *Comm. ACM*, v. 15, n. 10, p. 859-866, out. 1972.

DIMES, Troy. **Scrum Essencial**. Canadá: Babelcube Inc., 2014. Disponível em: <<http://doc.otsr.org/3.2/en/html/>>. Acessado em: 30 jun. 2015.

DOROW, Emerson. **O que é Governança de TI?** Disponível em: <<http://www.profissionaisti.com.br/2009/03/o-que-e-governanca-de-ti/>>. Acesso em: 1 out. 2015.

DOROW, Emerson. **O que é Governança de TI e para que existe?** 2010. Disponível em: <<http://www.governancadeti.com/2010/07/o-que-e-governanca-de-ti-e-para-que-existe/>>. Acesso em: 5 jul. 2015.

ELEMAR JR. BDD na prática – parte 1: **Conceitos básicos e algum código**. Disponível em: <<http://elemarjr.net/2012/04/11/bdd-na-prtica-parte-1-conceitos-bsicos-e-algun-cdigo/>>. Acesso em: 27 set. 2015.

ENGHOLM JR, Hélio. **Engenharia de software na prática**. São Paulo. Editora Novatec, 2010.

EXTREME PROGRAMMING. **A gentle introduction**. Agile Process. Disponível em: <<http://www.extremeprogramming.org/>>. Acesso em: 28 jun. 2015.

FALBO, R. A. **Engenharia de Software**: Notas de Aula. Universidade Federal do Espírito Santo. 2005. Disponível em: <<http://www.inf.ufes.br/~falbo/download/aulas/es-g/2005-1/NotasDeAula.pdf>>. Acesso em: 20 jul. 2015.

FATTOCS. FATTO Consultoria e Sistemas. **Glossário sobre Análise de Pontos de Função**. 2006. Disponível em: <www.fatto.com.br>. Acesso em: 17 ago. 2015.

Feature driven development and agile modeling. Agile Modeling. Disponível em: <<http://agilemodeling.com/essays/fdd.htm>>. Acesso em: 28 jun. 2015.

FERNANDES, Aguinaldo Aragon, ABREU, Vladimir Ferraz de. **Implantando a Governança de TI – da Estratégia à Gestão dos Processos e Serviços**. 3. ed. Rio de Janeiro: Brasport, 2012.

FERNANDES, D. B. **Metodologia dinâmica para o desenvolvimento de sistemas versáteis**. São Paulo: Érica, 1999.

FERREIRA, Gladstone. **Gerência de Configuração**. Disponível em: <<http://www.cin.ufpe.br/~gfn/qualidade/gc.html>>. Acessado em: 23 ago. 2015.

FILHO, Alberto Boller. **Artigo da Revista Engenharia de Software edição 24**. Disponível em: <<http://www.devmedia.com.br/artigo-engenharia-de-software-24-gerencia-de-configuracao/16805#ixzz3jksIWZdJ>>. Acesso em: 24 ago. 2015.

FILHO, Antônio M. S. **Gestão de Projetos de Software**. Disponível em: <<http://www.devmedia.com.br/gestao-de-projetos-de-software/9143>>. Acesso em: 16 ago. 2015.

FRANCISCANI, Juliana; PESTILI, Ligia C. **CMMI e MPS.BR**: um estudo comparativo. [S.1.], [S.1.]. Disponível em: <<http://www.unicerp.edu.br/images/revis-tascientificas/3%20-%20CMMI%20e%20MPS.BR%20Um%20Estudo%20Comparativo1.pdf>>. Acesso em: 27 set. 2015.

FREITAS, Marcos André dos Santos. **Fundamentos do gerenciamento de serviços de TI: preparatório para a certificação ITIL® V3 Foundation**. Rio de Janeiro: Brasport, 2010.

FUMSOFT. **Modelo MPS.BR**. Belo Horizonte, 2015. Disponível em: <http://www.fumsoft.org.br/qualidade/modelo_mpsbr>. Acesso em: 26 set. 2015.

FURTADO, T. B. **Evolução de Software**: fundamentos, processos e aplicação. 2007. 47f. Trabalho de conclusão de curso (Curso de Sistemas de Informação). Centro de Ensino Superior de Juiz de Fora. Rio de Janeiro, 2007. Disponível em: <http://repositorio.ufla.br/bitstream/1/5383/1/TCC_Evolu%C3%A7%C3%A3o%20de%20Software.pdf> Acesso em: 22 jul. 2015.

GIANESINI, D. **Pesquisa da Engenharia de Requisitos em Empresas de Desenvolvimento de Software de Micro, Pequeno e Médio Porte de Joinville**. 2008. 22f. Trabalho de Conclusão de Curso (Curso de Sistema de Informação) – Instituto Superior Tupy (Sociedade Educacional de Santa Catarina - SOCIESC), Joinville, 2008. Disponível em: <https://www.google.com.br/url?sa=t&rct=j&q=&esrc=s&source=web&cd=6&cad=rja&ved=0CGsQFjAF&url=http%3A%2F%2Fwww.cpgmne.ufpr.br%2Fredeticbr%2Findex.php%3Foption%3Dcom_phocadownload%26view%3Dcategory%26download%3D98%3A2008-2-pesquisa-da-engenharia-de-requisitos-em-empresas-de-desenvolvimento-de-software-de-micro-pequeno-e-mdio-porte-de-joinville-debora-gianesini%26id%3D7%3Aengenharia-de-software%26Itemid%3D89&ei=u-0YUpzmCKiB2gWs9YClAw&usg=AFQjCNG666bKh63JFWuMnY_hqHBDPrI7Sg&bvm=bv.51156542,d.eWU>. Acesso em: 29 jul. 2015.

GOMEDE, Everton. **Áreas de Conhecimento segundo o SWEBOK**. Disponível em: <<http://evertongomede.blogspot.com.br/2010/08/areas-de-conhecimento-segundo-o-swebok.html>>. Acesso em: 6 jul. 2015.

GRIEBLER, Fabricio. DDD, FDD, TDD, ATDD, BDD. **Que tal começar não confundindo as siglas?** Disponível em: <<http://blog.fasagri.com.br/?p=113>>. Acessado em: 27 set. 2015.

GROFFE, Renato J. **CMMI**: uma visão geral. DevMedia. São Paulo, [S.1.]. Disponível em: <<http://www.devmedia.com.br/cmmi-uma-visao-geral/25425>>. Acesso em: 26 jun. 2015.

GUEDES, Gilleanes T. A. **UML 2**: uma abordagem prática. 2. ed. São Paulo: Editora Novatec, 2011.

HAZAN, Claudia; LEITE, J. C. S. **Indicadores para a Gerência de Requisitos**. 2003. Disponível em: <http://wer.inf.puc-rio.br/WERpapers/pdf_counter.lua?wer=WER03&file_name=claudia_hazan.pdf>. Acesso em: 25 jul. 2015.

HEIMBERG, Viviane; GRAHL, Everaldo Artur. **Estudo de caso de aplicação de métrica de pontos de casos de Uso numa empresa de software**. XIII SEMINCO 2005. Seminário de computação. Disponível em: <<http://www.inf.furb.br/seminco/2005/artigos/130-vf.pdf>>. Acesso: 19 ago. 2015.

HEPTAGON. **O Que É FDD?** Disponível em: <<http://www.heptagon.com.br/fdd-oque>>. Acesso em: 26 set. 2015.

HIRAMA, K. **Engenharia de Software**: qualidade e produtividade com tecnologia. Rio de Janeiro: Elsevier, 2011.

HIRAMA, Kichi. **Engenharia de software**: qualidade e produtividade com tecnologia. Rio de Janeiro: Elsevier, 2012.

HONORATO, Icaro Henrique. **Modelo de Desenvolvimento Adaptativo de Software (DAS)**. Disponível em: <<https://prezi.com/qlglj1nmczj4/modelo-de-desenvolvimento-adaptativo-de-software-das/>>. Acesso em: 26 set. 2015.

IEEE. **IEE Standard for software reviews and audits**. 1988.

IMENES, Elison Roberto. **Seleção de Ferramentas CASE**. 2006. Monografia (Bacharelado em Ciência da Computação) – Curso de Ciência da Computação da Faculdade de Jaguariúna, Jaguariúna.

INSTITUTO DE SOFTWARE. **Comparação do MPS.BR com o CMMI**. São Paulo, 2015. Disponível em: <<http://its.org.br/o-que-e-mpsbr/comparacao-do-mps-br-com-o-cmmi>>. Acesso em: 26 set. 2015.

ISACA. Management Guidelines: **Information Systems Audit and Control Association & Foundation**. 2000.

ISO. **Standards**. Disponível em: <<http://www.iso.org/iso/home.htm>>. Acesso em: 20 set. 2015.

ISO/IEC 12207:2008(en). **Systems and software engineering**: software life cycle processes. Disponível em: <<https://www.iso.org/obp/ui/#iso:std:iso-iec:12207:ed-2:v1:en>>. Acesso em: 22 set. 2015.

ISO/IEC 9126. **The International Organization for Standardization and the International Electrotechnical Commission**. ISO/IEC TR 9126-4:2004, Software engineering - Product quality.

ITGI (2005). Disponível em: <<http://www.itgi.org/>>. Acesso em: 30 set. 2015.

ITGI. Information Security Governance. Guidance for Board of Directors and Executive Management. Second Edition 2006. Disponível em: <<http://www.itgi.org>>. Acesso em: 30 set. 2015.

ITIL. Disponível em <<http://www.itil-officialsite.com>>. Acesso em: jun. 2015.

KAN, Stephen – Metrics and Models en Software Quality Engineering – Addison-Wesley, Reading, 1995.

KERZNER, Harold. **Project management: a systems approach to planning, scheduling, and controlling.** 6th ed. New York: John Wiley & Sons Inc., 1998.

LAHOZ, Carlos; SANT' ANNA, Nilson. **Os Padrões ISO/IEC 12207 e ISO/IEC 15504 e a Modelagem de Processos da Qualidade de Software.** Disponível em: <http://mtc-m18.sid.inpe.br/col/lac.inpe.br/worcap/2003/10.20.14.01/doc/LahozWorkcap_versaofinal.pdf>. Acesso em: 22 set. 2015.

LEMBE DE VEIGA, Jurema Forinda. **Ferramentas para Gerência de Projetos - Engenharia de Software 25.** Disponível em: <<http://www.devmedia.com.br/ferramentas-para-gerencia-de-projetos-engenharia-de-software-25/17147#ixzz3e6hLO6N6>>. Acesso em: 22 jun. 2015

LOPES, Leandro Teixeira. **Um modelo de Processo de Engenharia de Requisitos para ambientes de desenvolvimento distribuído de Software.** 2004. 21f. Dissertação de Mestrado (Curso de Ciência da Computação) – Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, 2004. Disponível em: <http://tede.pucrs.br/tde_busca/arquivo.php?codArquivo=103>. Acesso em: 25 jul. 2015.

MACÊDO, Diego. **Gerenciamento dos custos do Projeto (PMBoK 5 Edição).** 21, abr. 2014. Disponível em: <<http://www.diegomacedo.com.br/gerenciamento-dos-custos-do-projeto-pmbok-5a-ed/>>. Acesso em: 30 jun. 2015.

MACHADO, F. N., **Análise e Gestão de Requisitos de Software, onde nascem os sistemas.** São Paulo: Érica, 2011.

MAGALHÃES, I. L.; PINHEIRO, W. B. **Gerenciamento de serviços de TI na prática: uma abordagem com base na ITIL.** São Paulo: Novatec, 2007.

MAGAZINE. Modelagem da gestão de pessoas: **Revista Engenharia de Software Magazine 55**. Disponível: <<http://www.devmedia.com.br/modelagem-da-gestao-de-pessoas-revista-engenharia-de-software-magazine-55/26916#ixzz3jLZtpcjQ>>. Acesso em: 17 ago. 2015.

MAGUIRE, Steve. **Guia Microsoft para o desenvolvimento de programas sem erros**. Rio de Janeiro: Campus, 1994.

MANIFESTOAGIL. **Manifesto para o desenvolvimento ágil de software**. Disponível em: <<http://www.manifestoagil.com.br/index.html>>. Acesso em: 27 set. 2015.

MARTINS, José Carlos Cordeiro. **Gerenciamento Projetos de Desenvolvimento de Software com PMI, RUP e UML**. 5. ed. Rio de Janeiro: Brasport, 2010.

MEDEIROS, Ernani. **Desenvolvendo Software com UML 2.0**. São Paulo: ed. Makron Books, 2004.

MEYER, Bertrand. **Object-Oriented Software Construction**. Addison-Wesley, 1997.

MINGAY, S; BITTINGER, S. **Combine CobiT and ITIL for Powerful IT Governance, in Research Note**. Gartner, 2002. Disponível em: <<http://www3.gartnet.com>>. Acesso em: jun. 2015.

MOLINARI, Leonardo. **Testes de software: produzindo sistemas melhores e mais confiáveis**. São Paulo: Érica, 2003.

MYERS, Glenford J. **The Art of Software Testing**. 2. ed. Nova Jérsei: John Wiley & Sons, 2004.

NASCIMENTO, Valéria Moura. Tema: **Gerência de riscos em planejamento e Controle de Projetos**. Monografia em Administração de Empresas pela Universidade Veita Almeida. Rio de Janeiro, 2003. p. Disponível em: <<https://www.uva.br/sites/all/themes/uva/files/pdf/monografia-gerenciamento-de-risco-em-projetos.pdf>>. Acesso em: 16 ago. 2015.

NETO, Arilo C. D. Artigo Engenharia de Software: **Introdução a Teste de Software**. Disponível em: <<http://www.devmedia.com.br/artigo-engenharia-de-software-introducao-a-teste-de-software/8035>>. Acesso em: 26 set. 2015.

OLIVEIRA FILHO, Carlos M. Kalibro: **interpretação de métricas de código-fonte**. Dissertação (Mestrado em Ciências) - Instituto de Matemática e Estatística/USP. São Paulo. 2013.

OLIVEIRA, Macelo N. De; LIMA, Iremar N. De. **Processo de desenvolvimento de software de acordo com a norma ISO/IEC 15504**. 2012. Disponível em: <<http://blog.newtonpaiva.br/pos/e5i19-processo-de-desenvolvimento-de-software-de-acordo-com-a-norma-isoiec-15504/>>. Acesso em: 23 set. 2015.

OLIVEIRA, Rafael Braga de. Framework FUNCTEST: **Aplicando** padrões de software na automação de testes funcionais. 2007. 109 p. (Dissertação de Mestrado - Universidade de Fortaleza - UNIFOR). Disponível em: <<http://docsslide.com.br/documents/dissertacao-559796f7a700f.html>>. Acessado em: 26 set. 2015.

PFLEGER, Shari Lawrence. **Engenharia de software, teoria e prática**. 2. ed. São Paulo: Pearson, 2004.

PINHEIRO, Flávio R. **Curso e-learning ITIL® Foundation V3**. edição 2011. Disponível em: <http://www.tiexames.com.br/curso_itil_v3_foundation.php>. Acessado em: 1 out. 2015.

PINTO, Evandro Moreira. **A Gestão de Requisitos como uma ferramenta de apoio à área comercial**. 2012. Disponível em: <<http://wm2info.com.br/blog/2012/01/17/a-gestao-de-requisitos-como-uma-ferramenta-de-apoio-a-area-comercial/>>. Acesso em: 28 jul. 2015.

PIRES, Eduardo. **DDD, TDD, BDD. Afinal o que são essas siglas?** Disponível em: <<http://eduardopires.net.br/2012/06/ddd-tdd-bdd/>>. Acesso em: 26 set. 2015.

PORTELLA, Cristiano R. R. **Tema da Aula:** Normas e Padrões de Qualidade de Software I. Disponível em: <http://www.cesarkallas.net/arquivos/faculdade/engenharia_de_software/17-Qualidade%20em%20Software/Normas%20de%20Qualidade-I.pdf>. Acessado em: 24 set. 2015.

PRESSMAN, R. S. **Engenharia de software**. 6. ed. São Paulo: McGraw-Hill, 2006, 720 p.

PRESSMAN, R. S. **Engenharia de software**: uma abordagem profissional. 7.ed. Porto Alegre: AMGH Editora Ltda., 2011.

PRESSMAN, Roger S. **Engenharia de software**. 6. ed. São Paulo: Makron Books, 2009.

PRESSMAN, Roger S. **Engenharia de software**. Mc Graw Hill, 6 ed, Porto Alegre, 2010.

PRESSMAN, Roger S. **Engenharia de Software**. São Paulo: Makron Books, 1995.

PRESSMAN, Roger S. **Engenharia de software**. 3. ed. São Paulo: McGraw-Hill, 2006.

PROJECT MANAGEMENT INSTITUTE (PMI). Um guia do conhecimento em gerenciamento de projetos: **Guia PMBOK. 5. ed.** Editora Saraiva. 2014.

PROJECT MANAGEMENT INSTITUTE. **PMI**. Disponível em: <<https://brasil.pmi.org>>. Acesso em: 20 ago. 2015.

PROJECT MANAGEMENT INSTITUTE. **PMI. Project Management Body of Knowledge – PMBoK**. 5. ed. Pennsylvania: USA, 2013.

PSMC. **ISO/IEC 15939**. 2015. Disponível em: <<http://www.psmsc.com/iso.asp>>. Acesso em: 24 set. 2015.

RAMOS, S. E; LAGARES, V. C; FERREIRA, R. E. P.; **Scrum no Teste de Software**. Disponível em: <http://www.omegabrasil.com.br/userfiles/produtos_categorias_14_catalogo.pdf>. Acesso em: 26 set. 2015.

REINALDO, Werley Teixeira; FILIPAKIS, Cristina D'Ornellas. **Estimativa de Tamanho de Software Utilizando APF e a Abordagem NESMA**. In: XI Encontro de Estudantes de Informática do Tocantins, 2009, Palmas. Anais do XI Encontro de Estudantes de Informática do Tocantins. Palmas: Centro Universitário Luterano de Palmas, 2009. p. 151-160. Disponível em: <<http://tinyurl.com/yhbftxb>>. Acesso em: 27 ago. 2015.

RIOS, Emerson; MOREIRA, Trayahú. **Teste de software**. 3^a ed. Rio de Janeiro: Alta Books, 2013.

ROCHA, Fabio Gomes. **Introdução ao desenvolvimento guiado por teste (TDD) com JUnit**. Disponível em: <<http://www.devmedia.com.br/introducao-ao-desenvolvimento-guiado-por-teste-tdd-com-junit/26559>>. Acesso em: 26 set. 2015.

RODRIGUES, Luiza. **Quais são os papéis do Scrum?** Disponível em: <<http://blog.myscrumhalf.com/2011/07/quais-sao-os-papeis-do-scrum-faq-scrum/>>. Acesso em: 22 set. 2015.

SANTOS, Jefferson de Barros. **Extraindo o melhor de xp, agile modeling e rup para melhor produzir software.** Disponível em: <<http://www.xispe.com.br/evento2002/download.html>>. Acesso em: 29 jun. 2015.

SAYÃO, Miriam; BREITMAN, Karin Koogan. **Gerência de Requisitos.** 2009. Disponível em: <http://www-di.inf.puc-rio.br/~karin/prominp/index_files/gerencia_req.pdf>. Acesso em: 26 jul. 2015.

SCHIASSATO, Jéssica e PEREIRA, Rodolfo. **TDD, DDD e BDD – Práticas de desenvolvimento.** Disponível em: <<http://www.princiweb.com.br/blog/programacao/tdd/tdd-ddd-e-bdd-praticas-de-desenvolvimento.html>>. Acesso em: 27 set. 2015.

SCHOFIELD, Joe; ARMEMTROUT, Alan; TRUJILLO, Regina. Pontos de função, Pontos de caso de uso, Pontos de história: observações de um estudo de caso. *CrossTalk – Jornal da engenharia de Software de defesa*, v. 26, n. 3, p. 23-27, junho 2013.

SILVA, Aloyana C. da; CORREA, Amanda dos Santos; SILVA, Isabella C. da, ROSA, Mariana; COSTA, Rogério C. da. **Segurança da Informação: Padrões de segurança da informação.** Universidade Estácio de Sá. 2015. Disponível em: <<https://ajeitarakoisa.files.wordpress.com/2015/07/si-padroes-de-seguranca.pdf>>. Acessado em: 24 set. 2015.

SILVA, F. Q. B. da; CÉSAR, A. C. F. **An Experimental Research on the Relationships between Preferences for Technical Activities and Behavioural Profile in Software Development.** *Software Engineering, 2009. SBES '09. XXIII Brazilian Symposium on*, p. 126-135, outubro 2009.

SILVA, Marcelo Gaspar Rodrigues; GOMEZ, Thierry Albert M. Pedroso; MIRANDA, Zailton Cardoso de. **TI: mudar e inovar: resolvendo conflitos com ITIL® v3: aplicado a um estudo de caso.** Brasília: Senac DF, 2010. 328 p.

SIMPOI 2008. **A Influência da Gerência de Requisitos na Retenção do Conhecimento em Empresas Desenvolvedoras de Software.** Artigo Científico. Grupo de Estratégia em Serviços, Inovação e Conhecimento - GESICON. Disponível em: <http://www.gesicon.com.br/uploads/2012/06/2008_00004.pdf>. Acesso em: 28 jul. 2015.

SINÉSIO Thiago. **Crystal Clear Methodologies.** Disponível em: <<http://pt.slideshare.net/ThiagoSinsio/metodologia-crystal-clear>>. Acessado em: 23 set. 2015.

SOARES, Michel dos Santos. **Comparação entre metodologias ágeis e tradicionais para o desenvolvimento de software**. Disponível em: <<http://www.dcc.ufla.br/infocomp/artigos/v3.2/art02.pdf>>. Acesso em: 26 jun. 2015.

SODRÉ, Elisângela B. **Mps Br – Melhoria do Processo de Software Brasileiro**. [S.1.], [S.1.]. Disponível em: <http://www.techoje.com.br/site/techoje/categoria/detalhe_artigo/245>. Acesso em: 26 set. 2015.

SOFTEX. Guia Geral de *Software*. **MPS.BR - Melhoria de Processo do Software Brasileiro**. 2012. Disponível em: <http://www.softex.br/mpsbr/_guias/guias/MPS.BR_Guia_Geral_Software_2012.pdf>. Acesso em: 28 jul. 2015.

SOFTEX. **MPS.BR**. [S.1.], 2014. Disponível em: <<http://www.softex.br/mpsbr/mps/mps-br-em-numeros>>. Acessado em: 26 jun. 2015.

SOFTWARE ENGINEERING INSTITUTE. **CMMI para Desenvolvimento – Versão 1.2**: Melhoria de processos visando melhores produtos. 2014. Disponível em: <http://www.sei.cmu.edu/library/assets/whitepapers/cmmi-dev_1-2_portuguese.pdf>. Acesso em: 26 jul. 2015.

SOMMERVILLE, I. **Engenharia de Software**. 6. ed. São Paulo: Addison Wesley, 2003.

SOMMERVILLE, Ian. **Engenharia de Software**. 9. ed. Pearson Ed Superior, 2011. 792 p.

SOMMERVILLE, Ian. **Engenharia de Software**. 9. ed. São Paulo. Pearson Addison Wesley, 2011. p. 59.

SOMMERVILLE, Ian. **Engenharia de Software**. Tradução Ivan Bosnic e Kalinka G.de O. Gonçalves; revisão técnica Kechi Hirama. 9. ed. São Paulo. Pearson Prentice Hall, 2011.

SOUZA, Givanaldo Rocha de. **Metodologias ágeis de desenvolvimento de software**. Disponível em: <<http://docente.ifrn.edu.br/givanaldorocha/disciplinas/engenharia-de-software-licenciatura-em-informatica/ESw03MetodologiasAgeis.pdf>>. Acesso em: 27 set. 2015.

THAMIEL, Thiago. **Entendendo Scrum**. Disponível em: <<http://thiagothamiel.wordpress.com/category/desenvolvimento-agil/page/2/>>. Acesso em: 2 fev. 2011.

TRENTIN, M. H. **Gerenciamento de projetos**. Guia para as certificações CAPM e PMP. 2. ed. Atlas, 2001.

TRISTACCI, Carlos. **CMMI**. Disponível em: <<http://www.carlostristacci.com.br/blog/cmmi/>>. Acesso em: 8 dez. 2013.

VARGAS, Ricardo Viana. Gerenciamento de projetos: **estabelecendo diferenciais competitivos**. 7. ed. Rio de Janeiro: Brasport, 2009.

VASCONCELOS A. M. L. de et al. **Introdução à engenharia de software e à qualidade de software**. UFLA/FAEPE, 2006. 57 p. (Curso de Pós-graduação “Lato Sensu” (Especialização) à Distância – Melhoria de Processo de Software.

VENEZIANI, Ana Cristine. **Norma ISO 9126**. Disponível em: <<http://usabilideiros.com.br/index.php/qualidade-de-software/item/5-norma-iso-9126>>. Acesso em: 24 set. 2015.

VERAS M. Cloud Computing: **Nova arquitetura de TI**, Rio de Janeiro, Brasport, 2012. CDD-004.678.

VERZELLO, Robert J.; REUTTER III, John. **Processamento de dados**. São Paulo: McGraw-Hill, 1984.

WAZLAWICK, Raul Sidnei. **Engenharia de software: conceitos e práticas**. Rio de Janeiro: Elsevier, 2013.