

Rosilene Fernandes

ENGENHARIA DE SOFTWARE

Tecnologia

ENGENHARIA DE SOFTWARE

Rosilene Fernandes





Engenharia de Software

Rosilene Fernandes



Ficha Catalográfica elaborada pela Fael. Bibliotecária – Cassiana Souza CRB9/1501

F363e Fernandes, Rosilene

Engenharia de software / Rosilene Fernandes. – Curitiba: Fael, 2017.

264 p.; il.

ISBN 978-85-60531-80-6

1. Engenharia de software 2. Computação I. Título

CDD 005.1

Direitos desta edição reservados à Fael.

É proibida a reprodução total ou parcial desta obra sem autorização expressa da Fael.

FAEL

Direção Acadêmica	Francisco Carlos Sardo
Coordenação Editorial	Raquel Andrade Lorenz
Revisão	Editora Coletânea
Projeto Gráfico	Sandro Niemicz
Capa	Vitor Bernardo Backes Lopes
Imagen da Capa	Shutterstock.com/Mclek
Arte-Final	Evelyn Caroline dos Santos Betim

Sumário

CARTA AO ALUNO | 5

1. INTRODUÇÃO À ENGENHARIA DE SOFTWARE | 7
2. REQUISITOS DE SOFTWARE | 31
3. PROJETO DE SOFTWARE | 55
4. CONSTRUÇÃO DE SOFTWARE | 77
5. TESTES DE SOFTWARE | 101
6. MANUTENÇÃO DE SOFTWARE | 123
7. GERENCIAMENTO DE CONFIGURAÇÃO DE SOFTWARE | 147
8. GERENCIAMENTO DE ENGENHARIA DE SOFTWARE | 171
9. FERRAMENTAS, MÉTODOS E TÉCNICAS NA ENGENHARIA DE SOFTWARE | 197
10. QUALIDADE DE SOFTWARE | 219

CONCLUSÃO | 243

GABARITO | 245

REFERÊNCIAS | 253

Carta ao Aluno

PREZADO(A) ALUNO(A),

APRESENTO O LIVRO de Engenharia de Software. Esta obra foi elaborada especialmente para que você possa compreender como essa disciplina pode lhe auxiliar ao longo de sua vida profissional dentro área da tecnologia da informação.

ESCREVER ESTE LIVRO foi um desafio e tanto, pois constantemente me preocupei em escrever de maneira clara, para que você acompanhasse meu pensamento. Espero que meu objetivo seja alcançado e que você curta o conteúdo do livro.

A engenharia de software possui uma quantidade imensa de assuntos, que não seria possível abordar totalmente nos capítulos deste livro. Assim, se você gostar da disciplina e quiser se aprofundar no assunto, pode consultar as referências citadas nesta obra.

É muito importante que você realize as atividades propostas em cada capítulo para fixar os conceitos discutidos. Em cada capítulo, você vai encontrar um tópico introdutório do conceito sobre o tema, uma seção sobre os assuntos específicos do tema, e uma síntese do capítulo antes das atividades propostas.

Lembre-se que a leitura completa do capítulo é muito importante para sua compreensão. No entanto, se ainda estiver com dúvida, realize pesquisas na internet e faça leituras complementares indicadas nas referências (isso o ajudará).

Fico na esperança de que o conteúdo deste livro possa contribuir para seu crescimento profissional e despertar o interesse pela área da Engenharia de Software.

Um grande abraço,

Professora Rosilene Fernandes¹.

1 Mestre em Informática na área de Engenharia de Software, especialista em Gestão de Projetos de Software, e bacharel em Análise de Sistemas. Possui mais de dez anos de experiência em tecnologia da informação e ministra disciplinas nos cursos de graduação e pós-graduação da área.

1

Introdução à Engenharia de Software

NA ATUAL SOCIEDADE da informação, observamos que cada vez mais e mais sistemas são controlados por softwares e muitas vezes a economia dos países desenvolvidos é dependente do bom funcionamento desses sistemas. Nesse contexto, a Engenharia de Software tem como principal finalidade viabilizar o desenvolvimento profissional de softwares por meio de técnicas que possibilitem o direcionamento de especificações, projetos e evoluções. O objetivo deste capítulo é apresentar os principais conceitos da Engenharia de Software e fornecer base para compreensão das responsabilidades de um engenheiro de software.

1.1 A Engenharia de Software e sua história

Entre os anos 1960 e 1970, o desenvolvimento de softwares apresentava muitas dificuldades, os projetos ultrapassavam o orçamento, os softwares desenvolvidos eram de baixa qualidade, os projetos não cumpriam os prazos e por muitas vezes não atingiam os requisitos esperados. Tais dificuldades eram estimuladas pelo rápido crescimento da demanda por softwares, da complexidade dos problemas a serem resolvidos e da inexistência de técnicas estabelecidas para o desenvolvimento de sistemas que funcionassem adequadamente. Assim, essa época ficou conhecida como “crise de software”.

Mediante esse cenário e em uma tentativa de controlar a crise com um tratamento mais sistemático (com a utilização de métodos, processos, técnicas e ferramentas) ao desenvolvimento de sistemas de software, o termo *Engenharia de Software* foi usado pela primeira vez como tema de conferência patrocinada pela Organização do Tratado do Atlântico Norte (Otan) no ano de 1968. O objetivo da conferência foi fazer que o desenvolvimento de software fosse tão bem-sucedido quanto a engenharia tradicional. É necessário compreender que algumas vezes o projeto de engenharia tradicional apresenta falhas, como a emblemática Ponte de Tacoma, no entanto, entendia-se que o número de insucesso era muito inferior aos dos projetos de softwares.

Exemplo da problemática da Ponte de Tacoma:

Em 1928, a Câmara de Comércio da cidade de Tacoma, estado de Washington, EUA, inicia o estudo de viabilidade técnica para a construção de uma ponte, de Tacoma até a península Gig Harbor. Seus mentores foram Clark H. Eldridge, engenheiro de pontes, e Lacey V. Murrow, engenheiro-chefe. Com o engenheiro Eldridge, a construção ficou orçada, inicialmente, em US\$ 11 milhões. Devido à Grande Depressão de 1929, os setores públicos *Washington State Department of Transportations* e *Federal Public Works Administration* solicitaram sua revisão. Como o sr. Eldridge estava de férias, coube ao engenheiro-chefe procurar outro projetista, entrando em cena o renomado engenheiro Leon S. Moisseiff (experiente em projetos famosos, dentre destes o projeto da ponte Golden Gate). Com este revisando o desenho original, o orçamento caiu para apenas US\$ 7 milhões.

Finalmente a ponte pénsil Tacoma Narrows, de aproximadamente 853 metros e 1.600 metros de extensão teve suas obras iniciadas em 1938. Durante a construção, técnicos, engenheiros e demais funcionários percebe-

ram que, ao ser atingida por correntes de vento não tão intensas, a estrutura tendia a oscilar transversalmente, por isso passou a ser chamada carinhosamente de Galloping Gertie (Cavalinho Galopante). Vários ensaios foram realizados com o objetivo de reduzir tais oscilações, contudo, nenhum chegou a ter plena eficácia.

No verão de 1940, a ponte foi aberta ao tráfego rodoviário e, por sua peculiaridade, logo tornou-se atração turística. O que para os engenheiros era um verdadeiro horror estrutural, para algumas pessoas dirigir ou observar uma grande estrutura como se estivesse em uma montanha-russa era um prazer indescritível. No dia 7 de novembro de 1940, a ponte sofreu com ventos de aproximadamente 70 quilômetros por hora e acabou desabando (LEET; UANG; GILBERT, 2014, p. 45-48).

Saiba mais

Confira o vídeo *Tacoma Narrows Bridge Collapse*, que mostra a ponte e explica o que aconteceu: <https://www.youtube.com/watch?v=IXyG68_caV4>.



“Engenharia de software é uma disciplina de engenharia relacionada com todos os aspectos da produção de software, desde os estágios iniciais de especificação do sistema até sua manutenção, depois que este entra em operação” (SOMMERVILLE, 2011). O sentido da palavra *engenharia* apresenta os conceitos de projeção, construção e manutenção.

1.2 A importância da Engenharia de Software

Como visto, a Engenharia de Software é uma área do conhecimento da computação voltada para a especificação, o desenvolvimento e a manutenção de sistemas de software aplicando tecnologias e práticas de gerência de projetos e outras disciplinas, objetivando organização, produtividade e qualidade. A aplicação da Engenharia de Software envolve o uso de processos de software que permitem ao engenheiro especificar, projetar, implementar e manter sistemas de software, avaliando e garantindo suas qualidades. Além disso, deve oferecer mecanismos para planejar e gerenciar o processo de desenvolvimento.

Empresas desenvolvedoras de software passaram a empregar esses conceitos sobretudo para orientar áreas de desenvolvimento, uma vez que a adoção destes traz vários benefícios para o projeto, como:

- × geração de artefatos concretos produzidos durante o desenvolvimento do software com o objetivo de ajudar a descrever as funcionalidades, a arquitetura e o design do software;
- × fornecimento de linguagem única, ou seja, linguagem para comunicação interpessoal, envolvendo a equipe técnica e os usuários do projeto;
- × estabelecimento de um ambiente no qual os membros da equipe absorvam os conhecimentos necessários de todo processo de desenvolvimento do software;
- × identificação dos papéis entre os participantes, com objetivo de definir as responsabilidades para cada um deles;
- × integração entre as diferentes equipes, nas quais os participantes compartilham tarefas;
- × compartilhamento de processos comuns em diversos projetos;
- × definição de critérios para certificação de todos os processos por meio de auditoria;
- × facilitação da participação de novos membros da equipe;
- × possibilidade de que qualquer indivíduo da equipe se afaste do projeto sem que o processo de desenvolvimento seja parado.

Esses benefícios deixam claro que a adoção de um processo para o desenvolvimento de qualquer sistema é essencial para o sucesso do projeto de software. Dentre desses processos, no tópico subsequente, serão discutidos os métodos tradicionais e os métodos ágeis.

1.3 O processo de software

Um processo de software é definido como um conjunto estruturado de atividades necessárias para o desenvolvimento de um produto (SOMMER-

VILLE, 2011). Pressman (2011) define um processo de software como uma forma base para o controle da gestão de projetos de software que determina o contexto no qual são aplicados métodos técnicos e gerados produtos derivados (modelos, documentos, dados, relatórios, formulários, entre outros), incluindo a definição de marcos em que a qualidade é assegurada e as mudanças são conduzidas adequadamente.

De maneira geral, o processo de software integra as atividades necessárias para o desenvolvimento do produto de software a ser entregue, incluindo sua documentação. Essas atividades, detalhadas na sequência, envolvem:

- ✖ **especificação de software**, que procura levantar e compreender a especificação do produto de software, no qual o engenheiro de software tem de compreender o problema, bem como a funcionalidade e o comportamento esperados para o sistema;
- ✖ **projeto de software**, no qual os requisitos tecnológicos são incorporados aos requisitos essenciais do sistema, possibilitando a definição da plataforma que será utilizada para a implementação;
- ✖ **implementação**, que realiza a produção do software, atendendo à especificação e ao projeto definido;
- ✖ **validação**, na qual diversos níveis de validações são realizados, desde validação unitária de cada componente do software, passando por validações de integração até a validação das funcionalidades do sistema; o objetivo dessa fase é garantir que o software atenda ao que o cliente deseja;
- ✖ **entrega e implantação**, responsável por colocar o software em produção; para isso, é necessário treinar os usuários e configurar o ambiente de produção;
- ✖ **evolução de software**, que ocorre após a implantação e a entrega dele; a evolução pode ocorrer porque é necessário fazer a correção de erros encontrados após a entrega, porque o software precisa ser adaptado para acomodar mudanças em seu ambiente externo ou porque o cliente necessita de funcionalidade adicional ou de aumento de desempenho.

Mesmo que as atividades do processo de software indiquem naturalmente uma ordem de precedência entre as atividades, isso não significa que se deve aguardar a conclusão de uma atividade em sua totalidade para partir para outra. Ou seja, as atividades do processo de software podem ser organizadas de maneiras distintas.

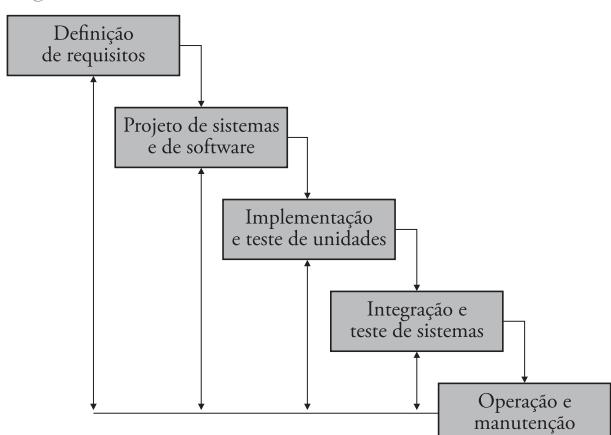
A fim de representar algumas estruturas de atividades do processo de software, são definidos modelos de processo do ciclo de vida do software. Segundo Pfleeger (2004), os modelos de processos podem se comportar de forma sequencial, na qual as atividades seguem determinada ordem; incremental, na qual ocorre a divisão de escopo; iterativa, na qual ocorre a retroalimentação das atividades; ou evolutiva, na qual o software é aprimorado.

1.3.1 O modelo em cascata: um modelo sequencial

Os modelos sequenciais se preocupam em organizar o processo em atividades sequenciadas linearmente. O principal modelo com essa característica é o modelo em cascata, a partir do qual diversos outros modelos foram propostos, inclusive modelos incrementais e evolutivos. Comumente chamado de modelo de ciclo de vida clássico, o modelo em cascata organiza as atividades do processo de software de forma sequencial, como demonstra a figura 1.1, em que cada fase envolve a elaboração de um ou de mais artefatos que devem ser aprovados antes de a fase seguinte ser iniciada.

Assim, uma fase só deve ser começada após a conclusão daquela que a antecede. Uma vez que, na prática, essas fases se sobreponem de alguma forma, geralmente, permite-se um retorno à fase anterior para a correção de erros. A entrega do sis-

Figura 1.1 - O modelo em cascata



Fonte: Sommerville (2011).

tema completo ocorre em um único momento, ao fim das fases de entrega e de implantação.

Pode-se dizer que o modelo em cascata é o modelo de ciclo de vida mais antigo, no entanto, críticas questionam sua eficiência, tais como as apresentadas por Pressman (2011):

- ✗ projetos reais muitas vezes não seguem o fluxo sequencial proposto pelo modelo;
- ✗ requisitos devem ser estabelecidos de maneira completa, correta e clara logo no início do projeto; a aplicação deve, portanto, ser entendida pelo desenvolvedor desde o início do projeto, entretanto, é difícil para o usuário colocar todos os requisitos explicitamente. O modelo em cascata requer isso e tem dificuldade de acomodar a incerteza natural que existe no início de muitos projetos;
- ✗ o usuário precisa ser paciente; uma versão operacional do software não estará disponível até o fim do projeto;
- ✗ a introdução de certos membros da equipe, tais como projetistas e programadores, é frequentemente adiada desnecessariamente; a natureza linear do ciclo de vida clássico leva a estados de bloqueio nos quais alguns membros da equipe do projeto precisam esperar que outros membros completem tarefas dependentes.

Embora o modelo apresente pontos críticos, Sommerville (2011) afirma que as vantagens do modelo em cascata consistem na documentação produzida em cada fase e sua aderência a outros modelos de processo de engenharia. Muitos outros modelos mais complexos são, na realidade, variações do modelo cascata, incorporando laços de realimentação (PFLEEGER, 2004).

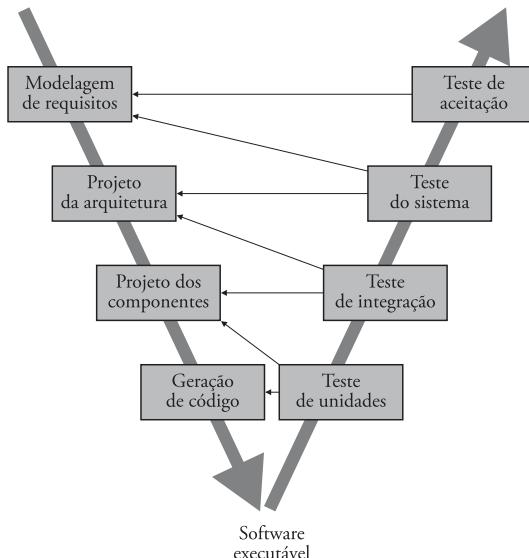
Assim, em princípio, o modelo em cascata deve ser usado apenas quando os requisitos são bem compreendidos e pouco provavelmente serão alterados durante o desenvolvimento do sistema (SOMMERVILLE, 2011).

1.3.2 O modelo em V: um modelo sequencial

O modelo em cascata possui algumas variações, como o modelo em V, apresentado na figura 1.2. De acordo com Pfleeger (2004), esse modelo

enfatiza a relação entre as atividades de teste: teste de unidade, teste de integração, teste de sistema e teste de aceitação, com as demais fases do processo de desenvolvimento.

Figura 1.2 - O modelo em V



Fonte: Adaptado de Pfleeger (2004).

O sentido do modelo é seguido de cima para baixo a partir do lado esquerdo; quando o código-fonte estiver concluído, é seguido de baixo para cima no lado direito, ou seja, quando existir um software executável efetivamente, subimos no modelo. Basicamente, não existe diferença entre o modelo em V e o modelo em cascata; o modelo V apenas enfatiza uma forma para visualizar como a verificação e as ações de validação são aplicadas.

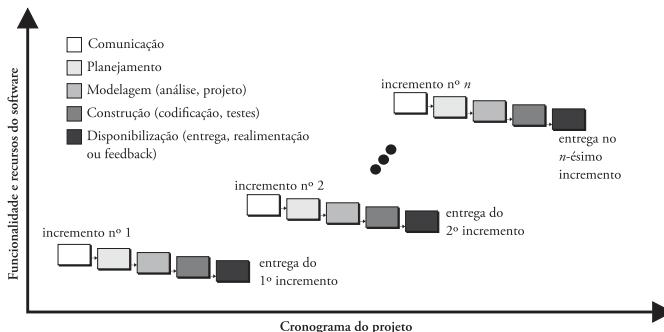
Os modelos sequenciais propõem a entrega completa do sistema, somente após a realização de todas as atividades do desenvolvimento. No entanto, nem sempre os clientes estão dispostos a esperar o tempo necessário para que isso ocorra, principalmente quando se trata de grandes sistemas (PFLEEGER, 2004). Dependendo do tamanho do sistema, pode-se demorar meses e até anos para que seja concluído, inviabilizando sua espera. Por esse motivo, outros modelos foram propostos visando à redução do tempo de

desenvolvimento. Nesse contexto, a entrega por versões possibilita ao usuário a utilização de algumas funcionalidades do sistema enquanto outras ainda estão sendo desenvolvidas.

1.3.2 O modelo incremental

Pressman (2011) explica que o modelo incremental libera uma série de versões, denominadas incrementos, que oferecem progressivamente maior funcionalidade ao cliente à medida que cada incremento é entregue. A ideia central do modelo é que, a cada ciclo ou iteração, uma versão operacional do sistema seja produzida e entregue para uso/avaliação do cliente. O primeiro incremento normalmente contém funcionalidades centrais, tratando os requisitos básicos, e as outras características são tratadas em ciclos subsequentes.

Figura 1.3 - O modelo incremental



Fonte: Adaptado de Pressman (2011).

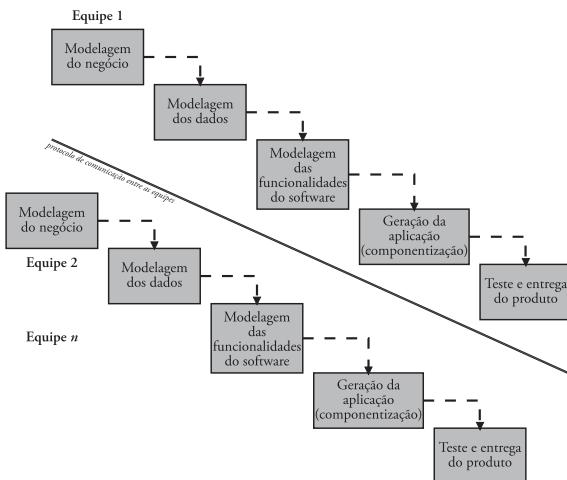
Dependendo do tempo estabelecido para a liberação dos incrementos, algumas atividades podem ser feitas para o sistema como um todo ou não. O modelo incremental é particularmente útil quando não há pessoal suficiente para realizar o desenvolvimento dentro dos prazos estabelecidos ou para lidar com riscos técnicos, tal como a adoção de uma nova tecnologia (PRESSMAN, 2011).

1.3.3 O modelo RAD: um modelo incremental

O RAD (*Rapid Application Development* ou Desenvolvimento Rápido de Aplicação, em português), apresentado na figura 1.4, é um modelo incre-

mental que enfatiza um ciclo de vida curto, aproximadamente entre 60 dias e 90 dias (PRESSMAN, 2011). É utilizado principalmente para aplicações em que os requisitos são bem compreendidos e o objetivo do projeto é restrito.

Figura 1.4 - O modelo RAD



Fonte: Adaptado de Pressman (2011).

- × **Modelagem do negócio:** o fluxo de informação entre as funções dos negócios é modelado. Nessa fase, devem ser respondidas as seguintes questões: Que informação dirige o processo do negócio? Que informação é gerada? Quem gera a informação? Para onde vai a informação? Quem processa a informação?
- × **Modelagem dos dados:** nessa fase, o fluxo de informação definido na modelagem dos negócios é refinado, as características de cada objeto são identificadas e as relações entre eles são estabelecidas.
- × **Modelagem das funcionalidades do software:** nessa fase, os objetos de dados são transformados para se obter o fluxo de informação necessário para implementar uma função do negócio e são criadas descrições do processamento para manipular esses objetos.

- ✖ **Geração da aplicação:** nessa fase, são utilizadas ferramentas automatizadas para facilitar a construção do software, as quais permitem o reuso de componentes de programas já existentes, quando possível, ou a criação de componentes reutilizáveis.
- ✖ **Teste e entrega do produto:** nessa fase, devido ao reuso, muitos componentes do programa já foram testados, o que reduz o tempo total de teste, entretanto, todo o restante deve ser testado.

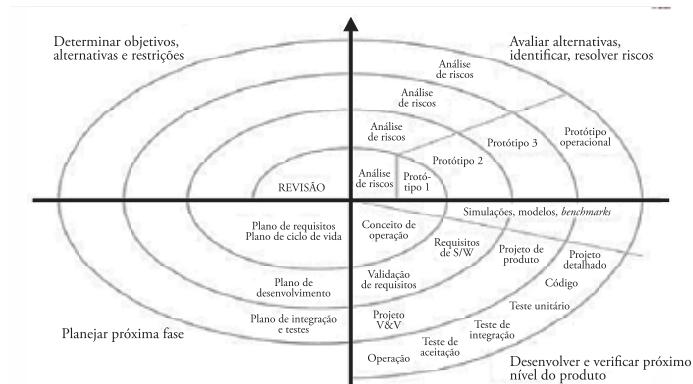
1.3.4 O modelo espiral: um modelo evolucionário

Como já discutido, os sistemas de software evoluem ao longo do tempo e seus requisitos podem mudar com frequência ao longo de seu desenvolvimento (PRESSMAN, 2011). Tal situação fortalece a existência de modelos de processo que lidem com essas incertezas, e é nesse contexto que os modelos evolucionários se encaixam.

Enquanto modelos incrementais têm por base a entrega de versões operacionais desde o primeiro ciclo, os modelos evolutivos não têm essa preocupação. Na maioria das vezes, os primeiros ciclos produzem protótipos ou apenas modelos. À medida que o desenvolvimento avança e os requisitos vão ficando mais claros e estáveis, protótipos dão lugar a versões operacionais até que o sistema completo seja construído. Assim, quando o problema não é bem definido e não pode ser totalmente especificado no início do desenvolvimento, é melhor optar por um modelo evolutivo. A avaliação ou o uso do protótipo/sistema pode aumentar o conhecimento sobre o produto e melhorar o entendimento que se tem acerca dos requisitos. Entretanto, a gerência do projeto e a gerência de configuração precisam ser bem estabelecidas e conduzidas.

O modelo em espiral é um modelo evolucionário, proposto por Boehm (1988), e apresenta um *framework* de processo de software dirigido a riscos. Nesse modelo, o processo de desenvolvimento de software é representado por uma espiral, sendo cada volta da espiral uma fase do processo, conforme apresentado na figura 1.5.

Figura 1.5 - O modelo espiral



Fonte: Sommerville (2011).

Dessa maneira, a volta mais interna pode se dedicar à viabilidade do sistema, seguida pela definição de requisitos, pelo projeto do sistema e assim por diante. Cada volta da espiral é dividida em quatro setores, como explica Sommerville (2011):

- × **determinar objetivos, alternativas e restrições** – os objetivos específicos da etapa são definidos, bem como as restrições ao processo e ao produto, e um plano de gerenciamento é elaborado;
- × **avaliar alternativas, identificar e resolver riscos** – para cada risco identificado no processo é feita uma avaliação e as medidas para redução de riscos são tomadas, como desenvolvimento de protótipos;
- × **desenvolver e verificar próximo nível do produto** – com a avaliação de riscos, um modelo de desenvolvimento é selecionado, por exemplo, prototipação descartável, caso os riscos sejam muito grandes;
- × **planejar próxima fase** – o projeto é revisado e uma nova fase da espiral é planejada, se houver pertinência.

O principal destaque do modelo espiral é o reconhecimento explícito do risco como parte do processo de desenvolvimento de software. Os riscos são consequência do planejamento realizado e dos objetivos traçados e, a partir de sua definição, a empresa de software pode realizar seu processo de forma a

incorporar a redução dos riscos no processo de desenvolvimento (SOMMERVILLE, 2011).

1.3.4.1 Prototipação

Constantemente, clientes têm em mente um conjunto geral de objetivos para um sistema de software, mas não conseguem identificar claramente os requisitos que o sistema terá de atender. A prototipação auxilia os engenheiros de software e os clientes a compreender melhor o que deverá ser construído. Por mais que tenha sido citada anteriormente no contexto do modelo espiral, a prototipação pode ser aplicada no contexto de qualquer modelo de processo.

Encontramos três maneiras distintas de se aplicar a prototipação: a primeira é fazer um protótipo em papel, ou mesmo no computador, que retrate a interação homem-máquina; a segunda é implementar uma funcionalidade que já está no escopo do software a ser desenvolvido; e a terceira é utilizar-se de um software já pronto que tenha parte das ou todas as funcionalidades desejadas.

Mesmo sendo eficiente para a compreensão dos requisitos do sistema, um dos maiores problemas do uso da prototipação é que muitas vezes o cliente acredita que o protótipo já é o software pronto ou em fase de término e começa a pressionar para que a entrega do software ocorra rapidamente.

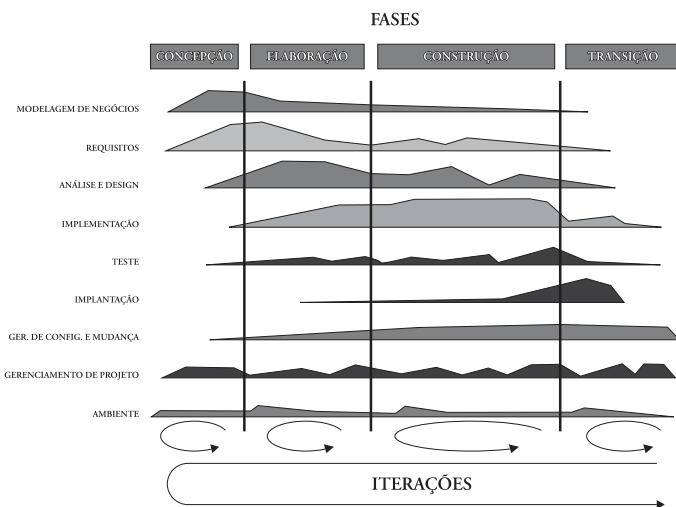
1.3.5 O modelo RUP: um modelo interativo e incremental

O RUP (*Rational Unified Process* ou Processo Unificado da Rational, em português) procura incorporar elementos dos modelos de processo anteriormente apresentados, em uma tentativa de incorporar as melhores práticas de desenvolvimento de software, dentre elas a entrega incremental (SOMMERVILLE, 2011). Seu objetivo é produzir softwares de alta qualidade que atendam às necessidades dos usuários com orçamento e cronograma controlados (KRUCHTEN, 2003).

No entanto, o RUP não é apenas um modelo de processo de software, é uma abordagem completa para o desenvolvimento de software, incluindo, além de um modelo de processo de software, a definição detalhada de responsabilidades/papéis, atividades, artefatos e fluxos de trabalho, entre outros. O

modelo de processo do RUP tem como diferencial principal a organização em duas dimensões, como ilustra a figura 1.6. Na dimensão horizontal é representada a estrutura do modelo em relação ao tempo, a qual é organizada em fases e iterações. Na dimensão vertical são mostradas as atividades, chamadas de disciplinas no RUP.

Figura 1.6 - Visão geral do RUP



Fonte: Adaptado de Kruchten (2003).

Sommerville (2011) descreve as quatro fases de negócio abordadas pelo RUP como:

1. **iniciação** – todas as entidades externas que interagirão com o sistema são descritas e as interações são definidas em *business cases*. Com isso, a avaliação de sistema para o negócio é feita, optando-se pela continuidade ou não do projeto;
2. **elaboração** – as metas dessa fase são o desenvolvimento da compreensão do desenvolvimento, o estabelecimento de um *framework* de arquitetura, a criação do plano do projeto e a identificação de riscos. No fim dessa fase deve existir um modelo de requisitos para o sistema.

3. **construção** – essa fase envolve projeto, programação e testes do sistema. As partes do sistema são desenvolvidas em paralelo e integradas. Ao fim da fase, o software deve estar funcional e a documentação para usuários, pronta.
4. **transição** – essa fase envolve a transferência do software da equipe de desenvolvimento para os usuários e sua operação em ambiente real. Ao fim dessa fase, o software deve funcionar no ambiente real.

Segundo Sommerville (2011), o RUP considera a existência de iterações no desenvolvimento de duas maneiras: cada fase pode ter sucessivas iterações, desenvolvendo-se os resultados de maneira incremental; e todo o conjunto de fases pode ser executado de maneira incremental.

1.4 Metodologias ágeis

Sem dúvida, um dos principais desafios do engenheiro de software é lidar com mudanças durante o desenvolvimento dos sistemas. Muitos profissionais de Tecnologia da Informação (TI) relatam que uma abordagem tradicional de desenvolvimento é inadequada para o desenvolvimento de sistemas sujeitos a mudanças constantes e a orçamentos limitados. Nesses casos, as metodologias ágeis são uma possibilidade.

De maneira geral, agilidade pode ser vista como capacidade de adaptação das organizações face às mudanças ocorridas no ambiente de negócios (SANTANA JÚNIOR, 2012). Uma equipe ágil é aquela que consegue responder rapidamente a mudanças, dá valor às características e às habilidades de cada membro e reconhece que a colaboração é a chave para o sucesso do projeto (PRESSMAN, 2011).

A iniciativa para criação das metodologias ágeis iniciou-se em 2001, quando um grupo de profissionais e pesquisadores de TI se uniu para delinear valores e princípios de desenvolvimento de software que viabilizariam, às equipes de desenvolvimento, produzi-lo rapidamente e responder às mudanças. Intitulando-se “aliança ágil”, os pioneiros da metodologia trabalharam por dois dias para chegar ao “manifesto da aliança ágil”, considerado oficialmente desde então como o início do movimento ágil.

Entende-se que nesse manifesto devem ser valorizados: os indivíduos e as interações mais que os processos e as ferramentas; o software em funcionamento mais que a documentação abrangente; a colaboração com o cliente mais que a negociação de contrato; e a resposta a mudanças mais que seguir um plano (PRESSMAN, 2011). Os princípios ágeis que compõem o manifesto são apresentados no quadro 1.1.

Quadro 1.1 – Princípios do manifesto ágil

Nº	Princípio
P1	Satisfazer ao cliente com entregas contínuas e frequentes
P2	Receber bem as mudanças de requisitos, mesmo em uma fase avançada do projeto
P3	Realizar entregas com frequência, sempre na menor escala de tempo
P4	Possibilitar que as equipes de negócio e de desenvolvimento trabalhem juntas diariamente
P5	Manter uma equipe motivada fornecendo ambiente, apoio e confiança necessários
P6	Viabilizar a maneira mais eficiente de a informação circular com uma conversa face a face
P7	Ter o sistema funcionando é a melhor medida de progresso
P8	Desenvolver processos ágeis que promovem o desenvolvimento sustentável
P9	Dar atenção contínua à excelência técnica e a um bom projeto para aumentar a agilidade
P10	Entender que simplicidade é essencial
P11	Saber que as melhores arquiteturas, requisitos e projetos provêm de equipes organizadas
P12	Possibilitar que, em intervalos regulares, a equipe reflita sobre como se tornar mais eficaz

Fonte: Adaptado de Beck (2001).

Uma metodologia é considerada ágil quando efetua o desenvolvimento de software de modo incremental, colaborativo, direto e adaptativo. O termo

incremental remete à ocorrência da disponibilização de pequenas versões em iterações de curta duração; já a expressão *colaborativa* refere-se a situações em que o cliente e os profissionais do software trabalham juntos, em permanente interação. A palavra *direta* indica que o método deve ser simples de se aprender e de se modificar. Por fim, *adaptativa* significa ser hábil em responder às mudanças até o último instante.

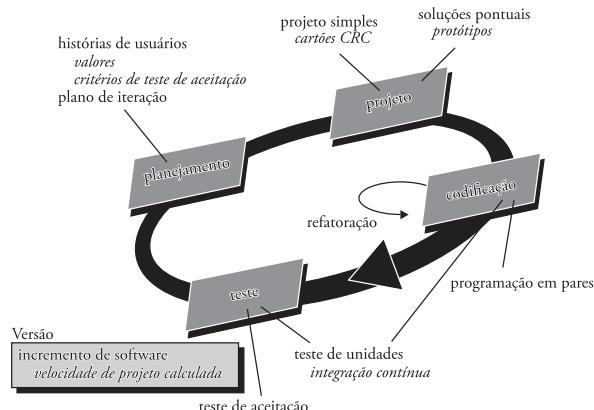
Nesse contexto, as metodologias ágeis que mais se destacam são: eXtreme Programming (XP), Scrum, Crystal, Feature Driven Development (FDD), Dynamic Systems Development Method (DSDM), Open Source Software Development. Entre as metodologias mais utilizadas, destaca-se o XP e o SCRUM.

1.4.1 Extreme programming (XP): uma metodologia ágil

O XP foi criado em meados de 1990 por Kent Bech no departamento de computação da montadora de carros Daimler Chrysler. Como uma metodologia ágil, o XP busca responder com velocidade às mudanças nas especificações do projeto de software.

A figura 1.7 apresenta o processo XP, destacando as fases: planejamento, projeto, codificação e teste.

Figura 1.7 – O processo XP



Fonte: Pressman (2011).

- ✗ **Planejamento:** as histórias de usuário são escritas e priorizadas pelo cliente e os membros da equipe estimam sua duração em semanas. Se a duração de uma história for maior do que três semanas, é solicitado ao cliente que divida a história e os clientes e os desenvolvedores decidem juntos quais histórias entrarão na próxima versão do sistema. Na entrega da primeira versão, a velocidade do projeto (quantidade de histórias implementadas) é calculada e a velocidade do projeto é usada para estimar datas de entrega futuras. Conforme o trabalho progride, o cliente pode adicionar, alterar, excluir ou dividir as histórias, a equipe XP aceita as mudanças e se replaneja.
- ✗ **Projeto:** nessa fase, simplicidade é a palavra de ordem. Parte-se do princípio de que um modelo simples é sempre preferível a um modelo complexo e que não se deve projetar funcionalidade extra assumindo que ela será necessária no futuro. Estruturas complexas tendem a se tornar cada vez mais caras e difíceis de manter. As histórias de usuários escritas na fase de planejamento devem ser transformadas em CRC (Classe, Responsabilidade e Colaboração) e cada CRC deve funcionar como um *script* que representa as funções que cada módulo do sistema desempenhará e como se relacionará com outros módulos. Os usuários devem ser envolvidos nessa fase e, ao ler o CRC, deve ser possível que uma pessoa consiga simular o papel do módulo que ele representa.
- ✗ **Codificação:** visto que o usuário participou de todo o planejamento, ele deve estar sempre disponível durante o processo de codificação para sanar dúvidas e colaborar com sugestões. O XP recomenda o uso de refatoração, que consiste na reorganização interna do código-fonte sem alteração no comportamento externo. Isso permite melhorias no projeto depois que a implementação já foi iniciada, uma vez que projeto e implementação ocorrem em paralelo. Outro ponto a destacar é a sugestão de que os desenvolvedores devem trabalhar em pares. No XP, a propriedade do código é coletiva, assim todos compartilham o mesmo orgulho e as mesmas críticas.

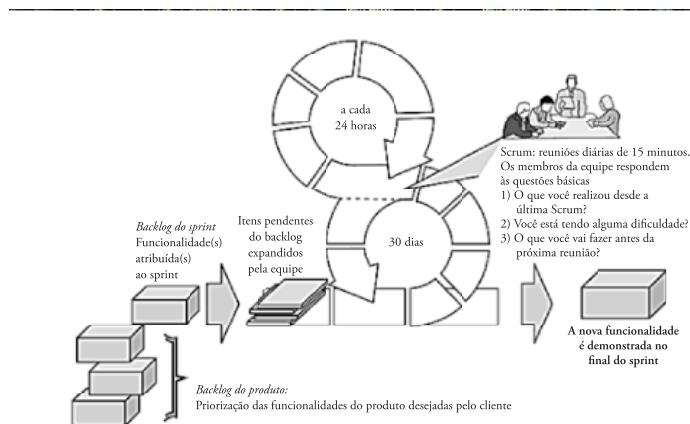
- ✗ **Testes:** os testes devem ser definidos e codificados antes mesmo que o módulo que será testado esteja escrito. Pensar nos testes que serão feitos promove um refinamento na codificação e elimina, de início, possíveis erros. Por fim, deve ser definido com os clientes um conjunto de regras que dirão que um módulo está pronto para entrar em produção, chamado testes de aceitação. Assim o usuário poderá comprovar que o módulo que foi desenvolvido corresponde a sua necessidade.

1.4.2 Scrum: uma metodologia ágil

O Scrum adota ideias da teoria de controle de processos industriais no desenvolvimento de softwares, reinserindo conceitos de flexibilidade, adaptabilidade e produtividade. O foco do método é encontrar uma maneira para que os profissionais de TI atuem de forma flexível para produzir o software em um ambiente de constantes mudanças. O nome dessa metodologia vem de uma atividade que ocorre em partidas de rúgbi. De acordo com Pressman (2011), os principais princípios de Scrum são:

- ✗ equipes pequenas organizadas para maximizar a comunicação, minimizar o *overhead* e compartilhar o conhecimento tácito e informal;
- ✗ processo adaptável a mudanças técnicas e de negócio;
- ✗ incrementos frequentes e regulares de software, que podem ser inspecionados, ajustados, testados, documentados e expandidos;
- ✗ trabalho e membros da equipe divididos em partições de baixo acoplamento;
- ✗ documentação e testes constantes feitos à medida que o produto é construído;
- ✗ processo capaz de declarar o produto como pronto “a qualquer momento, por qualquer motivo”.

Figura 1.8 – O fluxo do Scrum



Fonte: Pressman (2011).

O processo de Scrum envolve as atividades de levantamento de requisitos, análise, projeto, evolução e entrega. Como pode-se observar na figura 1.8, as tarefas de cada atividade são realizadas dentro de um padrão de processo chamado *sprint*, no qual uma lista priorizada de requisitos, dita *backlog*, é mantida. Os requisitos podem ser adicionados, removidos e alterados a qualquer momento, bem como as prioridades podem ser alteradas. Vale ressaltar que um *sprint* é uma unidade de trabalho com tempo pré-determinado (tipicamente 30 dias), durante o qual são escolhidos alguns requisitos do *backlog* considerados congelados, ou seja, não sujeitos a mudanças. Assim, a equipe pode trabalhar em um ambiente estável por um curto período de tempo.

Pequenas reuniões, de aproximadamente 15 minutos, são feitas diariamente pela equipe com o Scrum Master (o líder do projeto), na qual três perguntas-chave são feitas para cada membro da equipe: O que você realizou desde a última Scrum? Você está tendo alguma dificuldade? O que você vai fazer antes da próxima reunião?

Por fim, o produto resultante de um *sprint* é um *demo*, um incremento de software entregue ao cliente como demonstração para que seja avaliado e gere um *feedback*. Cada demo contém as funções até o último *sprint*.

1.5 Principais desafios da Engenharia de Software

A engenharia de software deve sempre pensar no desenvolvimento de softwares com independência funcional, baixo acoplamento e alta coesão, além de lidar com o aumento de diversidade, demandas pela diminuição do tempo para entrega e desenvolvimento de software.

- ✖ **Independência funcional:** o funcionamento não deve depender de outras partes, está ligado à modularidade, à ocultação de informações e à abstração. A ocultação de informações se dá pelo fato de não alocar dados sigilosos como senhas no mesmo *script* ou módulo.
- ✖ **Baixo acoplamento:** é a medida de interdependência entre dois ou mais módulos.
- ✖ **Alta coesão:** é a medida da força funcional entre módulos, isto é, quando um módulo disponibilizar dados, o outro que precisar poderá utilizá-los sem perdas, falhas ou necessidade de tratamento de dados.
- ✖ **Heterogeneidade:** é necessário que os sistemas de software operem com sistemas distribuídos e também com sistemas mais antigos (legados), sendo necessário aplicar técnicas para desenvolver sistemas flexíveis e confiáveis a fim de adaptar-se a essa heterogeneidade.
- ✖ **Entrega:** muitas técnicas tradicionais demandam tempo para obter a qualidade. O desafio da entrega consiste em reduzir os tempos de entrega dos sistemas grandes e complexos sem comprometer a qualidade.
- ✖ **Confiança:** os softwares estão presentes em todos os aspectos de nossa vida e precisamos confiar neles, sendo necessário desenvolver técnicas que demonstrem aos usuários que é possível confiar ter essa confiança.

1.6 Responsabilidade profissional do engenheiro de software

Os engenheiros de software devem se comportar de uma maneira honesta e eticamente responsável para serem respeitados como profissionais. O comportamento ético é mais do que simplesmente agir em concordância com a lei, envolve seguir um conjunto de princípios moralmente corretos, como:

- × respeito à **confidencialidade** de seus empregadores ou clientes, independentemente de haver ou não um acordo de confidencialidade formal assinado;
- × não falsear níveis de **competência** e aceitar trabalhos que estão fora de sua competência;
- × ter ciência das leis locais que regulam a propriedade intelectual, tais como patentes, direitos autorais, etc. e ser cuidadosos para assegurar que a **propriedade intelectual** dos empregadores e dos clientes esteja protegida;
- × não aplicar habilidades técnicas para uso indevido de computadores de outras pessoas, tendo em vista que a variação do **mau uso do computador** vai desde relativamente trivial (brincar com jogos na máquina de um empregador, por exemplo) a extremamente sérios (disseminação de vírus).

Preocupada que o engenheiro de software seja um profissional benéfico e respeitado, a Engenharia de Software possui código de ética e prática profissional, resultante dos esforços de uma equipe de trabalho liderada pelo IEEE (*Institute of Electrical and Electronics Engineers* ou Instituto de Engenheiros Eletricistas e Eletrônicos, em português), que foi criado em 1884 nos Estados Unidos e é uma sociedade técnico-profissional internacional dedicada ao avanço da teoria e da prática da engenharia nos campos da eletricidade, da eletrônica e da computação. Essa sociedade colabora no incremento da prosperidade mundial, promovendo engenharia de criação, desenvolvimento, integração, compartilhamento e conhecimento aplicado no que se refere à ciência e às tecnologias da eletricidade e da informação em benefício da humanidade e da profissão.

O código de ética e prática profissional da Engenharia de Software confere identidade à profissão, que deve ser utilizada pelos profissionais no sentido de desenvolver a cultura da responsabilidade dos engenheiros de software em relação à sociedade. Já a sociedade, por sua vez, está empenhada em obter qualidade nos produtos e nos serviços, o que se consolida na adoção de leis como o Código do Consumidor, criando uma cultura favorável à ética nos negócios.

O referido código de ética estabelece oito princípios para guiar o engenheiro de software:

- 1. público** – engenheiros de software devem agir de acordo com o interesse público;
- 2. cliente e empregador** – engenheiros de software devem agir para o melhor interesse de seu cliente e empregador e de acordo com o interesse público;
- 3. produto** – engenheiros de software devem garantir que produtos e modificações relacionadas atendam aos mais altos padrões profissionais possíveis;
- 4. julgamento** – engenheiros de software devem manter a integridade e a independência no julgamento profissional;
- 5. gestão** – gerentes e líderes de Engenharia de Software devem aceitar e promover uma abordagem ética para o gerenciamento de desenvolvimento e da manutenção de software;
- 6. profissão** – engenheiros de software devem aprimorar a integridade e a reputação da profissão de acordo com o interesse público;
- 7. colegas** – engenheiros de software devem auxiliar os colegas e ser justos com eles.
- 8. si próprio** – engenheiros de software devem participar da aprendizagem contínua durante toda a vida e devem promover uma abordagem ética para a prática da profissão.

O IEEE desenvolveu ainda um código de dez pontos de ética para seus membros, que requer que os membros se comprometam com a mais alta conduta profissional e ética, obrigados a serem honesto e rejeitarem a

corrupção em todas as suas formas, ajudar os colegas, evitar danos a outros, tratar todas as pessoas de forma justa e procurar, aceitar e oferecer crítica honesta de trabalho técnico. Na tomada de decisão, os membros do IEEE são obrigados a aceitar a responsabilidade na tomada de decisões consistentes com a segurança, a saúde e o bem-estar dos cidadãos. Para incentivar o cumprimento do Código de Ética IEEE, um prêmio anual para práticas éticas ilustres foi estabelecido.

Síntese

Foi visto neste capítulo que a Engenharia de Software surgiu da necessidade de se construir softwares com mais qualidade em menor tempo, sendo esta compreendida como uma disciplina de engenharia relacionada a todos os aspectos da produção de software.

Também foram apresentados os benefícios da adoção da engenharia de software, destacando e relatando os principais métodos tradicionais e os métodos ágeis da engenharia de software.

Por fim, foi ressaltado que os engenheiros de software possuem responsabilidades com a profissão e com a sociedade, devendo se preocupar com assuntos que vão além das questões técnicas.

Atividades

1. Por quem a engenharia de software deve ser executada? O que ela oferece?
2. Defina com suas palavras o que é um processo de software.
3. Compare os modelos sequenciais com os modelos incrementais e destaque a principal diferença entre eles.
4. Após ler sobre o código de ética e prática profissional da engenharia de software, explique como ele pode ajudar o engenheiro de software.

2

Requisitos de Software

A ÁREA DE requisitos de software trata de levantamento, análise, documentação e validação dos requisitos de software. A má condução dessas atividades torna projetos de engenharia de software criticamente vulneráveis. Requisitos de software expressam as necessidades e restrições colocadas sobre um produto de software que contribui para a solução de um problema do mundo real.

2.1 Requisitos de software

Comumente encontramos o termo “requisitos” em documentos acadêmicos, técnicos e livros da área de Tecnologia da Informação (TI), não é mesmo? Porém, muitas pessoas, ao lerem essa documentação pela primeira vez, não visualizam os requisitos como parte intrínseca do desenvolvimento do software, e sim como algo à parte, algo descolado do processo de desenvolvimento.

No entanto, isso não é uma verdade. O trabalho empregado no desenvolvimento dos requisitos faz parte da produção do software e desempenha um papel fundamental, sendo considerado um fator determinante para o sucesso ou o fracasso de um projeto de software. Observe algumas citações que reforçam esse pensamento:

a parte mais difícil da construção de software é decidir o que construir. Nenhuma outra etapa do trabalho conceitual é tão difícil quanto a fixação dos requisitos técnicos detalhados, incluindo todas as interfaces com usuários finais, com máquinas e outros sistemas. Nenhuma outra etapa compromete tanto o projeto se executada erroneamente. Portanto, a função mais importante que o construtor de software executa para seu cliente é a extração iterativa e o refinamento dos requisitos do produto (Frase de Fred Brooks, publicada em 1986, p. 1069–1076).

Não há nada tão inútil quanto fazer eficientemente o que não deveria ser feito (Frase de Peter Drucker, segundo Shore, 2014).

Existem diferentes definições para o termo “requisitos”. A seguir estão listadas algumas delas.

- × Requisitos de um sistema são descrições dos serviços que devem ser fornecidos por esse sistema e as suas restrições operacionais (SOMMERVILLE, 2011).
- × Um requisito de um sistema é uma característica do sistema ou a descrição de algo que o sistema é capaz de realizar para atingir seus objetivos (PFLEINGER, 2004).
- × Um requisito é uma propriedade que deve ser exposta para resolver algum problema do mundo real (SWEBOK, 2004).

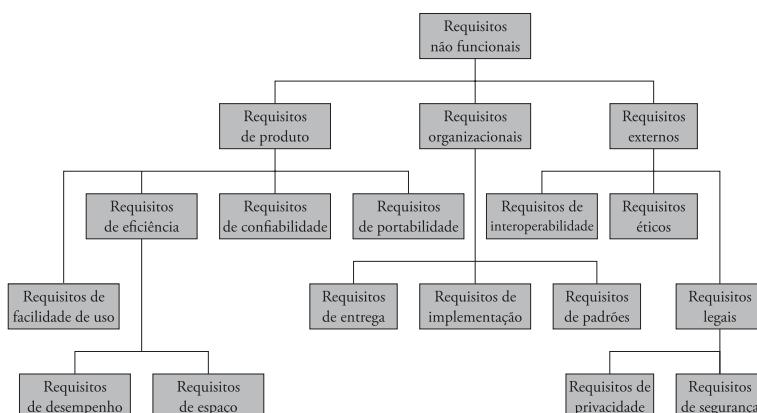
Dessa maneira, podemos afirmar que os requisitos de software expressam necessidades e restrições de um produto de software que contribui para a solução de um problema real. Na área de conhecimento dos requisitos de software, podemos encontrar diferentes tipos de requisitos, como: funcionais, de domínio e não funcionais.

- × **Requisitos funcionais:** descrevem funções que o software deve executar. Para Sommerville (2011), eles são declarações de serviços que o sistema deve prover, delineando o que o sistema deve fazer e, ainda, podem descrever como o sistema deve reagir a entradas específicas, como deve se comportar em situações específicas e o que não deve fazer (por exemplo, o sistema deve permitir inclusão, alteração e remoção de cadastro de funcionários com os seguintes atributos: nome, endereço, cidade, etc.)
- × **Requisitos de domínio ou regras de negócio:** são derivados do domínio do sistema a ser desenvolvido e refletem características e restrições desse domínio. Eles podem restringir requisitos funcionais existentes ou estabelecer como regras de negócio devem ser realizadas, refletindo fundamentos do domínio do sistema.
- × **Requisitos não funcionais:** são aqueles que agem para restringir uma solução. Segundo Sommerville (2011), eles descrevem restrições sobre os serviços ou funções oferecidos pelo sistema. Tais restrições são originadas por intermédio das necessidades dos usuários (restrições de orçamento, políticas organizacionais, necessidades de interoperabilidade com outros sistemas de software ou hardware ou em fatores externos como regulamentos e legislações) (SOMMERVILLE, 2011). Dessa maneira, os requisitos não funcionais podem ser classificados quanto à sua origem. Observe as classificações sugeridas por Sommerville (2011) e apresentadas na figura 2.1.
 1. **Requisitos de produto:** especificam o comportamento do produto de software. Referem-se a atributos de qualidade que o software deve apresentar, como facilidade de uso, eficiência (desempenho e espaço), confiabilidade e portabilidade (por exemplo, toda comunicação necessária entre o

ambiente e o usuário deve ser expressa no conjunto padrão de caracteres ANSI).

2. **Requisitos organizacionais:** são derivados de metas, políticas e procedimentos das organizações. Contêm requisitos de entrega (tempo para entrada no mercado, restrições de cronograma), requisitos de implementação (linguagem de programação, paradigma a ser adotado), requisitos de padrões (padrões de processo e modelos de documentos que devem ser usados) (por exemplo, o processo de desenvolvimento de sistema e os documentos a serem entregues deverão estar de acordo com o processo e os produtos a serem entregues definidos em XYZ).
3. **Requisitos externos:** referem-se a todos os requisitos derivados de fatores externos ao sistema e seu processo de desenvolvimento. Incluem requisitos de interoperabilidade com sistemas de outras organizações, requisitos éticos e requisitos legais (por exemplo, requisitos de privacidade e normas legais dos órgãos regulamentadores ANVISA, ANS, FEBRABAN, entre outros) (outro exemplo: o sistema não deverá revelar aos operadores nenhuma informação pessoal sobre os clientes).

Figura 2.1 – Tipos de requisitos não funcionais



Fonte: Sommerville (2011).

Como normalmente os sistemas dispõem de vários interessados, seus requisitos devem ser escritos de modo a serem compreendidos pelos diversos envolvidos: clientes, fornecedores, usuários, desenvolvedores, entre outros. Observe que cada envolvido pode ter um tipo de interesse sobre o sistema, por exemplo: os desenvolvedores se interessam pelos detalhes técnicos, já os clientes necessitam descrições mais abstratas, uma visão mais global do sistema. Assim, é importante apresentar os requisitos em diferentes níveis de descrição. Sommerville (2011) sugere dois níveis de descrição de requisitos:

- ✖ **requisitos de usuário** – utilizados para declarar os requisitos abstratos de alto nível, ou seja, de fácil compreensão pelos usuários do sistema que não têm conhecimentos técnicos. Normalmente eles são declarações em linguagem natural ou em diagramas sobre as funções que o sistema deve fornecer e as restrições sob as quais deve operar. Devem ser escritos para os envolvidos no sistema que não tenham um conhecimento detalhado do seu funcionamento;
- ✖ **requisitos de sistema** – utilizados para indicar a descrição detalhada do que o sistema deverá fazer. Portanto, estabelecem detalhadamente as funções e as restrições de sistemas. O documento de requisitos de sistema, algumas vezes chamado de especificação funcional, deve ser preciso e pode servir como um contrato entre o comprador do sistema e o desenvolvedor de software. Devem ter como alvo os profissionais técnicos de nível sênior e os gerentes de projeto.

2.2 Processo de requisitos

O processo de requisitos integra o processo de produção de software propondo métodos, técnicas e ferramentas que facilitem o trabalho necessário para a definição do que se quer de um software. Pressman (2011) destaca que as atividades do processo de requisitos são adaptadas às necessidades do projeto e tentam definir o que o cliente deseja, estabelecendo uma fundação sólida para o projeto e a construção do software.

Por mais que as atividades do processo de requisitos sejam adaptáveis para diferentes projetos e empresas, podemos tomar como base de estudos

que existem cinco grandes fases envolvidas nesse processo: (1) levantamento de requisitos, (2) análise de requisitos, (3) documentação de requisitos, (4) verificação e validação de requisitos e (5) gerência de requisitos.

2.2.1 Levantamento de requisitos

Segundo Pressman (2011), o levantamento de requisitos (também chamado de elicitação de requisitos) é o primeiro passo dentro do processo de requisitos. Realizado de maneira incorreta, poderá causar dano à qualidade do software, atrasar a entrega do produto e até mesmo provocar gastos fora do previsto.

O propósito do levantamento é coletar informações dos clientes, usuários e outras pessoas que tenham algum tipo de interesse no software, capturando essas informações como requisitos. Para estimular uma abordagem colaborativa, os envolvidos trabalham em conjunto para identificar o problema a ser tratado, propor elementos da solução, negociar diferentes abordagens (como atendimento de diferentes plataformas – web, mobile, internet) e especificar um conjunto preliminar de requisitos da solução (PRESSMAN, 2011).

O levantamento de requisitos é atividade complexa e envolta por diversas dificuldades, uma vez que engloba pessoas com diferentes conhecimentos, objetivos e até mesmo de gerações e culturas diferentes.

Carvalho (2001) apresenta uma lista das principais dificuldades encontradas nessa etapa.

- ✗ Falta de conhecimento do usuário de suas reais necessidades e do que o produto de software pode oferecer (item destacado também na figura 2.2).
- ✗ Falta de domínio do problema por parte do engenheiro de software.
- ✗ Domínio do processo de levantamento de requisitos pelos engenheiros de software.
- ✗ Comunicação inadequada entre os engenheiros de software e os usuários.
- ✗ Dificuldade do usuário de tomar decisões.

- ✖ Problemas comportamentais entre usuários e engenheiros de software no processo de levantamento de requisitos devido à ambiguidade nos papéis desempenhados por eles.
- ✖ Questões técnicas – as tecnologias de software e hardware mudam rapidamente, e os requisitos baseiam-se em conhecimentos detalhados ao domínio do usuário e alta complexidade de software.

Figura 2.2 – Levantamento de requisitos



Fonte: Shutterstock.com/AntartStock/Yusiki.

Existem várias técnicas que podem ser utilizadas no levantamento de requisitos. Podemos inclusive utilizar mais de uma técnica, principalmente quando existem diferentes focos de investigação. Entre as diferentes técnicas existentes, vamos conhecer o uso da entrevista, do questionário, da brainstorming, da JAD, da prototipagem, do workshop, da observação e da análise de documentos.

2.2.1.1 Entrevista

Essa técnica visa extrair informações importantes, por meio de encontros com os usuários. Deve-se ter o cuidado com perguntas erradas, ambíguas, não relevantes e que cansem o entrevistado. Os tipos de perguntas a serem usados são as abertas dirigidas, que abrangem mais o detalhamento de

um ponto, porém gastam mais tempo, e as fechadas, que focam mais na objetividade, porém oferecem problemas como a falta de detalhes e monotonia (CARVALHO, 2001).

Um entrevistador competente pode ajudar a entender e explorar os requisitos do produto de software. A correta aplicação requer habilidades sociais gerais, capacidade para escutar e conhecimento de táticas de entrevistas (ANDRIANO, 2006).

Segundo Lowdermilk (2013), é necessário transformar os pedidos abstratos dos usuários em necessidades significativas. Nesse contexto, o autor referencia a importância do UX (*User Experience*), termo frequentemente usado para resumir toda a experiência de um produto. Abrange não só a funcionalidade, mas também o quanto o produto é envolvente e agradável de usar.

A literatura define quatro fases para a realização de entrevistas:

1. **identificação de candidatos** – deve existir uma seleção criteriosa dos candidatos a serem entrevistados, já que um candidato mal selecionado pode resultar em uma perda de tempo ou no levantamento de dados incorretos;
2. **preparação** – é a etapa anterior à entrevista;
3. **desenvolvimento da entrevista** – no momento que começa a entrevista, o entrevistador deve fazer a introdução e explicar o objetivo da entrevista e quais suas metas. À medida que o entrevistado vai respondendo às perguntas, devem-se explorar as respostas por meio de resumos e obtenção de confirmação;
4. **continuação** – antes do início da próxima entrevista, deve-se realizar um resumo, depois enviar uma cópia ao entrevistado e pedir confirmação do resumo realizado.

De acordo com Kotonya (1998), existem dois tipos de entrevista: (i) entrevistas fechadas, em que o engenheiro de requisitos lança as perguntas para um universo já estudado de questões; e (ii) entrevistas abertas, em que não há agenda predefinida e o engenheiro de requisitos interage com os usuários sobre a definição do sistema.

As principais vantagens dessa técnica são: a possibilidade de contato direto com os atores que detêm conhecimento sobre os objetivos do software e a possibilidade de validação imediata por processos de comunicação. As principais desvantagens são: o problema do conhecimento tácito e as diferenças de cultura entre entrevistado e entrevistador.

2.2.1.2 Questionário

Técnica rápida que prevê obter informações de uma grande quantidade de usuários. Em sua elaboração, deve-se ter cuidados como: utilizar perguntas claras, objetivas e específicas (respostas podem ser antecipadas para dar mais velocidade ao processo), fazer perguntas mais importantes primeiro e agrupar perguntas sobre assuntos semelhantes (facilitando a compreensão do respondente). Algumas regras são importantes para apoiar o processo de construção de questionários.

- ✖ Deve ser definida a amostra, ou seja, o grupo populacional ao qual será aplicado o instrumento de obtenção de dados; a amostra condiciona a técnica ou técnicas de coleta de dados utilizadas e as características do próprio questionário.
- ✖ Existe concordância em que se deve partir de questões gerais para específicas.
- ✖ As perguntas mais concretas devem preceder as mais abstratas.
- ✖ Questões acerca de comportamento devem ser colocadas antes de questões acerca de atitudes.
- ✖ Devem ser colocadas primeiro as questões de caráter impessoal e só depois as de caráter pessoal.
- ✖ É necessário evitar ao máximo o chamado “efeito de contágio”, ou seja, a influência da pergunta precedente sobre a seguinte.
- ✖ As primeiras questões de descontração do respondente são chamadas de “quebra-gelo”, porque têm a função de estabelecer contato, colocando-o à vontade.
- ✖ O desenho visual de um questionário deve ser atrativo e facilitar o preenchimento das questões na sequência correta.

- ✗ Se o formato for muito complexo, os respondentes tendem a evitar questões, fornecer dados incorretos ou mesmo recusar-se a responder ao questionário.
- ✗ O número de questões introduzidas deve ser levado em conta: se forem em número excessivamente reduzido, podem não abranger toda a problemática que se pretende inquirir; se, pelo contrário, forem demasiadamente numerosas, arrisca-se ser de análise sobre os inquiridos, aumentando a probabilidade de não resposta.
- ✗ Um questionário deve parecer curto.
- ✗ Um questionário não deve ser demasiado longo.
- ✗ O questionário também não deve ser demasiado curto.
- ✗ Não se deve separar a mesma questão por páginas diferentes.
- ✗ As questões devem ser numeradas de maneira conveniente.
- ✗ Deve ser deixado espaço suficiente entre os itens.
- ✗ Quando são incluídas questões de resposta aberta, o espaço de resposta deve ser suficiente.
- ✗ Determinadas partes mais importantes devem ser sublinhadas ou destacadas.
- ✗ As instruções de preenchimento devem ser claramente distinguidas das questões e respostas alternativas.
- ✗ A identificação do respondente deve ser feita preferencialmente no início do questionário.
- ✗ Solicitação de cooperação: é importante motivar o respondente por meio de uma prévia exposição sobre a entidade que está promovendo a pesquisa e sobre as vantagens que a pesquisa poderá trazer.
- ✗ Instruções: as instruções apresentadas devem ser claras e objetivas.
- ✗ Os dados de classificação do respondente devem estar no final do questionário.
- ✗ O pesquisador deve fazer reflexões sobre as questões.
- ✗ Deve ser proporcionada ao respondente uma situação de liberdade.

Após a aplicação do questionário, é necessária a realização de um processo de análise, em que os resultados são tabulados para que se conheçam as limitações desse instrumento (BASTOS, 2005). Nesse processo de análise, recomenda-se que seus respondentes pertençam à população-alvo da pesquisa e tenham tempo suficiente para responder a todas as questões, e os engenheiros de software devem ser experientes.

No processo de análise, segundo Bastos (2005), com relação aos elementos funcionais do questionário, devem-se verificar a clareza e a precisão dos termos utilizados, a necessidade eventual de desmembramento das questões, a forma e a ordem das perguntas, a introdução e deve-se fazer uma reflexão sobre o valor de cada pergunta.

Segundo Leite (1994), questionários são utilizados quando se tem um bom conhecimento sobre a aplicação e se quer varrer um grande número de clientes. São importantes para que se possa ter uma ideia mais definida de como certos aspectos são percebidos por um grande número de pessoas e, se bem planejados, possibilitam análises estatísticas.

2.2.1.3 *Brainstorming*

É uma técnica para geração de ideias, em que se reúnem várias pessoas que fazem a sugestão de ideias sem que sejam criticadas ou julgadas, ou seja, as pessoas que participam desse tipo de reuniões sugerem e exploram suas ideias livremente. Existe um líder que conduz o brainstorming (VIANNA et al, 2012). Os principais benefícios são: fornece maior interação social, não limita o problema, há ausência da crítica e ajuda na eliminação de certas dificuldades do processo. Porém, como não é um processo estruturado, pode não obter os resultados esperados. O brainstorming tem duas fases:

- ✖ **geração das ideias** – os participantes fornecem ideias sem serem criticados, é necessário encorajar ideias irrelevantes, o número de ideias deve ser grande, etc.;
- ✖ **consolidação** – as ideias são discutidas, organizadas, revisadas e algumas são descartadas.

Geralmente, as reuniões se realizam com quatro a dez pessoas. Uma delas deve atuar como líder para começar, mas não para restringir. Envolve tanto a ideia de geração como a da redução de ideias. As ideias mais criativas e inovadoras geralmente surgem da combinação de ideias relacionadas.

O *brainstorming* apresenta algumas regras: não é permitida a crítica e o debate, deve-se permitir que a imaginação voe, é necessário gerar a maior quantidade de ideias possíveis, pode-se mudar e combinar as ideias, é possível reduzir as ideias, as ideias que não valem a pena discutir devem ser descartadas, é necessário agrupar as ideias similares em tópicos, por fim é importante priorizar as ideias.

2.2.1.4 JAD

Joint Application Development (JAD) é uma marca registrada da IBM. É um agrupamento de ferramentas destinadas a apoiar o desenvolvimento de sistemas de informação nas fases de levantamento de dados, modelagem e análise. Essas fases são realizadas pelos analistas de sistemas em conjunto com os usuários da aplicação.

A técnica JAD é uma ferramenta de negociação, visto que as sessões JAD para a definição de objetivos, metas, prioridades, custos e alocação de recursos são sempre realizadas juntamente com os usuários. Para Young (2002), JAD é um método para desenvolvimento de requisitos de software em que os representantes do usuário e os representantes do desenvolvimento trabalham juntamente com um facilitador para produzir uma especificação de exigência comum com a qual ambos concordem.

É uma técnica que provê comunicação, entendimento e trabalho em equipe entre usuários e desenvolvedores, facilitando assim a criação de uma visão compartilhada do que o produto possa ser. A técnica JAD visa criar sessões de trabalho estruturadas por meio de uma dinâmica de grupo e recursos visuais, em que analistas e usuários trabalham juntos para projetar um sistema, desde os requisitos básicos até o leiaute de telas e relatórios, prevalecendo a cooperação e o entendimento. Os desenvolvedores ajudam os usuários a formular os problemas e explorar possíveis soluções, envolvendo-os e fazendo com que eles se sintam participantes do desenvolvimento.

A técnica JAD tem quatro princípios: (1) dinâmica de grupo – utiliza sessões de grupo aumentando a capacidade dos indivíduos; (2) uso de técnicas visuais – gera aumento da comunicação e entendimento; (3) manutenção do processo organizado e racional – mantém o processo organizado; (4) utilização de documentação-padrão – devidamente preenchida e assinada por todos em uma sessão.

A técnica JAD tem seis participantes: (1) líder da sessão, responsável pelo sucesso do esforço; (2) engenheiro de requisitos, participante diretamente responsável; (3) executor, responsável pelo produto; (4) representantes dos usuários, pessoas que utilizarão o produto de software na empresa; (5) representantes do produto de software, pessoas bastante familiarizadas com as capacidades dos produtos de software; (6) especialista, pessoa que pode fornecer informações detalhadas sobre um tópico específico.

2.2.1.5 Prototipagem

Técnica que começa pelo estudo preliminar dos requisitos do usuário e é concluída com uma série de requisitos formulados, além de um protótipo que simula o comportamento do sistema. Essa técnica utiliza um processo interativo, o qual permite a definição do comportamento do sistema, e em particular sua aparência. De acordo com Davis (1992), um protótipo é uma implementação parcial de um sistema construído expressamente para aprender mais sobre um problema ou uma solução para um problema. Para Moule (2012), a prototipagem faz você se concentrar nos detalhes envolvidos em uma dada tarefa.

Desse modo, um protótipo de software implementa parte dos requisitos para compreender os requisitos que realmente importam para os usuários, além de investigar caminhos alternativos que poderiam satisfazer esses requisitos utilizando-se de mecanismos interativos. Podemos encontrar dois tipos distintos de protótipos de software na literatura: (1) descartável e (2) evolucionário.

1. O protótipo descartável é criado com o propósito de apresentar aos usuários o que foi capturado em relação aos requisitos do produto. Na criação de um protótipo descartável, o essencial é a rapidez de construção: um protótipo descartável deve ser produzido em poucas horas, ou no máximo em poucos dias. Esse protótipo tem por objetivo explorar aspectos críticos dos requisitos de um produto, auxiliando a decisão sobre questões que sejam importantes para o sucesso do produto.
2. O protótipo evolucionário deve conter um subconjunto dos requisitos do produto final, iniciando pelos requisitos de interface e evoluindo para a produção de um protótipo de software totalmente funcional.

Para Sommerville (2011), desenvolver um protótipo significa trabalhar diretamente com os requisitos. Nesse formato, o cliente interage com a equipe técnica apresentando as funcionalidades desejadas de acordo com sua necessidade.

2.2.1.6 Workshops

Essa técnica tem por objetivo reunir as partes interessadas de um projeto por um período curto de tempo para discutirem importantes aspectos do desenvolvimento da aplicação. Essa técnica é conduzida por um membro da equipe de projeto ou externo e necessita de uma preparação mais trabalhada do que uma reunião, além de envolver logística, distribuição do material com antecedência e sensibilização, envolvendo também os benefícios da sessão.

2.2.1.7 Observação

A técnica de observação baseia-se no mundo social ordenado. A ordem social é obtida sobre uma base de momento a momento por meio de ações coletivas dos participantes, no seu ambiente natural. Por exemplo, a ordem social será somente observável no caso em que o observador se submeta a ela. A observação é uma técnica muito utilizada, na qual o engenheiro de software procura ter uma posição passiva observando o ambiente em que o software irá atuar. Geralmente, essa técnica permite ao engenheiro de software fazer anotações sobre os objetos e vocabulários observados.

2.2.1.8 Análise de documentos

Técnica de levantamento tradicional que consiste na identificação de requisitos por meio da análise de documentos, manuais envolvidos no processo, análise de informações e/ou estudos de mercado. Essa técnica contempla a pesquisa e a busca de informação na documentação existente. A documentação inclui formulários, relatórios, manuais do sistema, manuais de políticas e diretrizes, contratos e documentos fiscais. Geralmente, a análise de documentos é utilizada em combinação com outras técnicas.

Como vimos, existem diversas técnicas para levantamento de requisitos. No entanto, a seleção deve considerar os seguintes pontos:

- × quais requisitos são conhecidos ou não, o que pode dinamicamente mudar em toda a vida do projeto;

- ✖ características do domínio do problema, o que é geralmente estático em toda a vida do projeto;
- ✖ características do domínio da solução, o que provavelmente muda toda vez que um tipo de solução para o problema for proposto;
- ✖ características do projeto, o que provavelmente muda toda vez que mudar a cultura ou a gerência.

Quadro 2.1 – Comparação das técnicas para levantamento de requisitos

Técnica	Pontos positivos	Pontos de atenção
Entrevistas	A possibilidade de contato direto com os atores que detêm o conhecimento sobre os objetivos do software e a possibilidade de validação imediata por processos de comunicação.	O problema do conhecimento tácito e as diferenças de cultura entre entrevistado e entrevistador.
Questionários	A padronização das perguntas e possibilidade de tratamento estatístico das respostas.	A limitação do universo de respostas e a pouca iteração, visto a impessoalidade da técnica.
<i>Brainstorming</i>	Fornece maior interação social, não limita o problema, há ausência da crítica e ajuda a tirar certas dificuldades do processo.	Algumas vezes, as ideias são apresentadas de maneira confusa, tornando difícil o refinamento, o desenvolvimento e a avaliação.
	Gera uma ampla variedade de pontos de vistas, estimula o pensamento criativo, constrói uma imagem mais completa do problema, provê um ambiente social muito mais confortável e mais fácil de aprender.	Pode ocorrer também de alguns participantes se manifestarem a favor ou contra a algumas ideias apresentadas, inibindo a produção do grupo.

Técnica	Pontos positivos	Pontos de atenção
JAD	Provê comunicação, entendimento e trabalho em equipe entre usuários e desenvolvedores, facilitando assim a criação de uma visão compartilhada do que o produto possa ser.	Quanto mais complexo o sistema, mais reuniões são necessárias. Todos os participantes devem estar familiarizados com as técnicas que serão utilizadas durante as reuniões.
Prototipagem	Permite alterar o sistema mais cedo no desenvolvimento, adequando-o mais de perto às necessidades do usuário, e a interação com o usuário.	Normalmente, várias iterações são necessárias para refinar um protótipo.
	Permite descartar um sistema quando este se mostrar inadequado.	Considerar o protótipo como sendo o sistema final: a qualidade pode não ter sido apropriadamente considerada.
Observação	Baixo custo e pouca complexidade da tarefa.	A dependência do ator (engenheiro de software) desempenhando o papel de observador e a superficialidade decorrente da pouca exposição ao universo que está sendo observado.
Análise de documentos	Facilidade de acesso e volume de informações.	Dispersão das informações e volume de trabalho.

2.2.2 Análise de requisitos

Após o levantamento dos requisitos, conforme apresentado na figura 2.2, é iniciada a fase de análise desses requisitos. Essa análise visa o completo entendimento das necessidades dos usuários, tendo como resultado os

requisitos descritos. A fase preocupa-se com o problema, não com soluções técnicas necessárias.

Para obter uma compreensão maior do sistema a ser desenvolvido, modelos de análise são construídos, os quais podem representar diferentes aspectos do sistema, como aspectos estruturais e comportamentais.

- ✖ **Aspectos estruturais:** preocupa-se com conceitos e interações que são importantes para o sistema em desenvolvimento. Fornece uma visão estática das informações de que o sistema necessita tratar.
- ✖ **Perspectiva comportamental:** preocupa-se com o comportamento geral do sistema e fornece uma visão desse comportamento.

Para Pressman (2011), os modelos criados durante a atividade de análise de requisitos ajudam a entender a informação, a função e o comportamento do sistema, tornando a tarefa de análise de requisitos mais fácil e sistemática. Tornam-se o ponto focal para a revisão e, portanto, a chave para a determinação da consistência da especificação.

Na análise de requisitos, é possível entender e detalhar os requisitos levantados. Seu objetivo é estabelecer um conjunto acordado de requisitos completos, consistentes e sem ambiguidades, que possa ser usado como base para as demais atividades do processo de desenvolvimento de software (KOTONYA; SOMMERVILLE, 1998).

Segundo Pfleeger (2004), análise atende a dois propósitos principais: (1) prover uma base para entendimento e concordância entre clientes e desenvolvedores sobre o que o sistema deve fazer e (2) prover uma especificação que guie os desenvolvedores nas demais etapas do desenvolvimento, sobretudo no projeto, na implementação e nos testes do sistema.

Usuários, clientes, especialistas de domínio e engenheiros de requisitos devem discutir os requisitos que apresentam problemas, negociar e chegar a uma concordância sobre as modificações a serem feitas. As discussões devem ser guiadas pelas necessidades da organização, incluindo o orçamento e o cronograma disponíveis. Porém, muitas vezes, as negociações são influenciadas por considerações políticas, e os requisitos são definidos em razão da posição e da personalidade dos indivíduos, não de argumentos e razões (KOTONYA; SOMMERVILLE, 1998).

Quando discussões informais entre analistas, especialistas de domínio e usuários não forem suficientes para resolver os problemas, é necessária a realização de reuniões de negociação, que envolvem (KOTONYA; SOMMERVILLE, 1998):

- × **discussão** – os requisitos que apresentam problemas são discutidos e os interessados presentes opinam sobre eles;
- × **priorização** – requisitos são priorizados para identificar requisitos críticos e ajudar nas decisões e no planejamento;
- × **concordância** – soluções para os problemas são identificadas, mudanças são feitas e um acordo sobre o conjunto de requisitos é acertado.

2.2.3 Documentação de requisitos

Durante a fase de levantamento e análise de requisitos, a equipe está focada em entender o problema a ser resolvido, ao que o sistema deve atender. Já na fase de documentação, ocorre a descrição textual desses requisitos, em formato de documento.

Segundo Easterbrook (2000), um bom documento de requisitos fornece vários benefícios: (i) facilita a comunicação dos requisitos; (ii) reduz o esforço de desenvolvimento, pois sua preparação força usuários e clientes a considerar os requisitos atentamente, evitando retrabalho nas fases posteriores; (iii) fornece uma base realística para estimativas; (iv) fornece uma base para verificação e validação; (v) facilita a transferência do software para novos usuários e/ou máquinas; (vi) serve como base para futuras manutenções ou incremento de novas funcionalidades.

Sommerville (2011) lembra que a documentação dos requisitos tem um vasto conjunto de interessados com diferentes interesses, como:

- × **clientes, usuários e especialistas de domínio** – são interessados na documentação de requisitos, visto que atuam na descrição, na avaliação e na alteração de requisitos;

- ✖ **gerentes de cliente** – utilizam o documento de requisitos para planejar um pedido de proposta para o desenvolvimento de um sistema, contratar um fornecedor e para acompanhar o desenvolvimento do sistema;
- ✖ **gerentes de fornecedor** – utilizam a documentação dos requisitos para planejar uma proposta para o sistema e para planejar e acompanhar o processo de desenvolvimento;
- ✖ **desenvolvedores** – utilizam a documentação dos requisitos para compreender o sistema e as relações entre suas partes;
- ✖ **testadores** – utilizam a documentação dos requisitos para projetar casos de teste, sobretudo testes de validação do sistema.

É importante alertar que não existe um padrão definido quanto à documentação de requisitos. Cada organização pode definir o seu padrão. Algumas decidem ter apenas um documento de requisitos, contendo diversas seções, outras fazem uso de vários documentos separados, um para requisitos funcionais, outro para requisitos não funcionais, e assim por diante. A seguir vamos observar um modelo básico para documentação de requisitos, apenas como exemplificação de como poderia ser um modelo de documentação de requisitos. No entanto, é possível encontrar outros modelos propostos pelo IEEE Std 830 (1998) e Robertson e Volere (2009).

Figura 2.3 – Modelo básico para documentação de requisitos

Requisitos funcionais

Identificador	Nome	Prioridade	Depende de
<RF01>	<Nome do requisito>		
Descrição	<Breve descrição do requisito>		
Especificação	<Passos necessários para atendimento do requisito>		
Histórico de revisão			
Alterações	Responsável		

Prioridade: 1 – alta, 2 – média, 3 – baixa

Requisitos não funcionais

Identificador	Nome	Tipo	Prioridade	Depende de
<RNF01>	<Nome do requisito>			
Descrição	<Breve descrição do requisito>			
Especificação	<Passos necessários para atendimento do requisito>			
Histórico de revisão				
Alterações	Responsável			

Prioridade: 1 – alta, 2 – média, 3 – baixa / Tipo: P – produto, O – organizacional, E – externo

Fonte: Elaborada pelo autor.

2.2.4 Verificação e validação de requisitos

A fase da validação de requisitos ocorre após a documentação dos requisitos, conforme vimos na figura 2.2. Essa fase tem como objetivo mostrar que os requisitos realmente definem o sistema que o cliente deseja. Ela é muito próxima da análise de requisitos, uma vez que se preocupa em descobrir problemas nos requisitos. Contudo, esses são processos distintos, já que a validação deve se ocupar da elaboração de um esboço completo do documento de requisitos, enquanto a análise envolve trabalhar com requisitos incompletos (SOMMERVILLE, 2011).

A validação de requisitos é importante, uma vez que a ocorrência de erros em um documento de requisitos pode levar a grandes custos relacionados ao retrabalho, quando esses erros são descobertos durante o desenvolvimento ou depois que o sistema estiver em operação. Como já comentado, o custo de fazer uma alteração no sistema, resultante de um problema de requisito, é muito maior do que reparar erros de projeto e de codificação.

Sommerville (2011) propõe que diferentes tipos de verificação sejam realizados sobre os requisitos no documento de requisitos.

- ✗ **Verificações de validade:** um usuário pode considerar que um sistema é necessário para realizar certas funções. Contudo, mais estudos e análises podem identificar funções adicionais ou diferentes,

que são exigidas. Os sistemas têm diversos usuários com necessidades diferentes e qualquer conjunto de requisitos é inevitavelmente uma solução conciliatória da comunidade de usuários.

- ✖ **Verificações de consistência:** os requisitos em um documento não devem ser conflitantes, ou seja, não devem existir restrições contraditórias ou descrições diferentes para uma mesma função do sistema.
- ✖ **Verificações de completeza:** o documento deve incluir requisitos que definam todas as funções e restrições exigidas pelos usuários do sistema.
- ✖ **Verificações de realismo:** utilizando o conhecimento da tecnologia existente, os requisitos devem ser verificados, a fim de assegurar que eles realmente podem ser implementados, levando-se também em conta o orçamento e os prazos para o desenvolvimento do sistema.
- ✖ **Facilidade de verificação:** para diminuir as possíveis divergências entre cliente e fornecedor, os requisitos do sistema devem sempre ser escritos de modo que possam ser verificados. Isso implica na definição de um conjunto de verificações para mostrar que o sistema entregue cumpre com esses requisitos.

Atualmente, encontramos diferentes técnicas de validação de requisitos, e essas podem ser utilizadas individualmente ou em conjunto.

- ✖ **Revisões de requisitos:** os requisitos são analisados sistematicamente por uma equipe de revisores, a fim de eliminar erros e inconsistências.
- ✖ **Geração de casos de teste:** como modelo ideal, os requisitos deveriam ser testáveis. Se os testes para os requisitos são criados como parte do processo de validação, isso, muitas vezes, revela problemas com os requisitos. Se um teste é difícil ou impossível de ser projetado, isso frequentemente significa que os requisitos serão de difícil implementação e devem ser reconsiderados.

Para Sommerville (2011), a validação de requisitos não deve ser subestimada, uma vez que, é muito difícil demonstrar que um conjunto de requisitos atende às necessidades de um usuário. A validação não consegue descobrir

todos os problemas com os requisitos, o que muitas vezes implica na necessidade de modificações para corrigir essas falhas de compreensão.

2.2.5 Gerência de requisito

Durante todo o desenvolvimento do software, pode haver necessidade de mudança, não importa o quanto a equipe tenha sido cuidadosa durante o levantamento, a análise, a documentação e a validação dos requisitos. Isso torna complexa a gerência dos requisitos, uma vez que a mudança de requisitos pode implicar no tempo de implementação ou até mesmo em mudanças em implementações já realizadas e testadas.

Dessa maneira, é necessário construir uma estrutura de requisitos que seja adaptável à mudanças e usar vínculos de rastreabilidade para representar as dependências entre os requisitos e outros artefatos do ciclo de vida do desenvolvimento. O gerenciamento de mudança inclui atividades como: determinar quais dependências são importantes para serem rastreadas, estabelecer a rastreabilidade, entre itens correlatos, e o controle de mudança.

Para garantir uma abordagem consistente, recomenda-se que as organizações definam um conjunto de políticas de gerência de mudança, contemplando (SOMMERVILLE, 2011):

- × processo de solicitação de mudança e as informações necessárias para processar cada solicitação;
- × processo de análise de impacto e de custos da mudança, além das informações de rastreabilidade associadas;
- × membros que formalmente avaliarão as mudanças;
- × ferramentas que auxiliarão o processo.

Síntese

Neste capítulo, foi visto que os requisitos devem estabelecer o que o software deve fazer e definir também as restrições de seu funcionamento e sua implementação.

Os requisitos podem ser classificados em funcionais, de domínio e não funcionais. Todos os requisitos, independentemente da sua classificação, devem ser definidos de maneira clara para que não haja problemas na sua interpretação, visto que é a partir da definição desses requisitos que o sistema será construído. O custo de fazer uma alteração no sistema, resultante de um problema de requisito, é muito maior do que reparar erros de projeto e de codificação.

Por fim, para auxiliar todo esse processo, as etapas a seguir são de suma importância.

- ✗ **Levantamento de requisitos:** tem o propósito de coletar informações de clientes, usuários e outras pessoas que tenham algum tipo de interesse no software, capturando essas informações como requisitos.
- ✗ **Análise de requisitos:** visa o completo entendimento das necessidades dos usuários, tendo como resultado os requisitos descritos. Essa fase preocupa-se com o problema, não com soluções técnicas necessárias.
- ✗ **Documentação de requisitos:** ocorre a descrição textual desses requisitos, em formato de documento.
- ✗ **Verificação e validação de requisitos:** tem como objetivo mostrar que os requisitos realmente definem o sistema que o cliente deseja.
- ✗ **Gerência de requisitos:** procura minimizar as dificuldades impostas pelas mudanças nos requisitos.

Atividades

1. Correlacione os itens a seguir.

(a)	Requisitos de sistema	()	Preocupa-se com o aprendizado e o entendimento das necessidades dos usuários e patrocinadores do projeto.
(b)	Levantamento de requisitos	()	Descrevem restrições sobre os serviços ou funções oferecidos pelo sistema.

(c)	Rastreabilidade	()	São descrições dos serviços que devem ser fornecidos pelo sistema e as suas restrições operacionais.
(d)	Requisitos não funcionais	()	Está ligada aos relacionamentos entre os requisitos.
(e)	Especificação de requisitos	()	Deve ser escrito em um formato compreensível por todos os interessados.

2. Identifique se os requisitos a seguir são do tipo funcional ou não funcional.
 - a) O sistema deve permitir cadastrar o cliente.
 - b) O sistema deve emitir um recibo para o cliente no ato do aluguel de um carro.
 - c) O sistema deve transformar um carro disponível em carro emprestado, quando o carro for alugado pelo cliente.
 - d) O sistema deve permitir cadastrar o cliente rapidamente, em menos de dois minutos.
 - e) O sistema deve emitir um recibo para o cliente, no ato do aluguel de um carro, com o tempo máximo de oito segundos após a transação.
3. Os requisitos não funcionais podem ser classificados quanto à sua origem. Descreva quais são essas classificações.
4. Escreva a quem interessa a documentação dos requisitos.

3

Projeto de software

PROJETO DE SOFTWARE é o processo de definição da arquitetura, dos componentes, das interfaces e de outras características de um sistema ou componente e também o resultado desse processo. Na etapa do projeto de software, os requisitos definidos na etapa anterior devem servir de referência para a obtenção da representação do software.

O OBJETIVO DESTE capítulo é apresentar os principais aspectos do projeto de software e fornecer base para a compreensão das principais técnicas de projeto existentes.

3.1 Fundamentação do projeto de software

Projetar um software é bastante complexo e grande parte dessa complexidade está associada à natureza mutável do software. Constantemente, softwares sofrem mudança e a maior pressão por elas vem da necessidade de modificação da funcionalidade do sistema que é incorporado pelo software.

Para Sommerville (2011), projeto de software é a descrição da estrutura do software que será implementado. Fortalecendo essa afirmação, Pressman (2011) define projeto de software como a representação significativa de alguma coisa que será construída. Assim, podemos entender que projetar um software significa determinar como os requisitos funcionais devem ser implementados na forma de estruturas de software.

No capítulo anterior, vimos que é necessário levantar, analisar e especificar um conjunto de requisitos para a construção do software. Nesse contexto, Pressman (2011) defende que o projeto de software é a única maneira com a qual se consegue traduzir com precisão os requisitos do cliente em um software ou sistema finalizado. Basicamente, o objetivo do projeto de software é incorporar tecnologia aos requisitos do usuário, projetando o que será construído na implementação; para tanto, é necessário conhecer a tecnologia disponível, bem como as facilidades e as dificuldades do ambiente de software no qual o sistema será implementado.

De maneira geral, o projeto de software tem início com um modelo de requisitos e esse modelo deve ser transformado em quatro níveis de detalhes:

1. **projeto de dados** – tem por objetivo projetar a estrutura de armazenamento de dados necessária para implementar o software;
2. **projeto arquitetural** – visa definir os grandes componentes estruturais do software e seus relacionamentos;
3. **projeto da interface** – descreve como o software deverá se comunicar dentro dele mesmo (interfaces internas), com outros sistemas (interfaces externas) e com pessoas que o utilizam (interface com o usuário);
4. **projeto como componente** – tem por objetivo refinar e detalhar a descrição dos componentes estruturais da arquitetura do software

Por fim, é produzida uma especificação do projeto composta por modelos de projeto que descrevem os dados, a arquitetura, as interfaces e os componentes. Em cada estágio, todos os itens da especificação são revistos quanto a clareza, correção, completeza e consistência com os requisitos e uns com os outros.

Segundo Pressman (2011), a importância do projeto de software pode ser estabelecida com uma única palavra: qualidade. Durante o processo de desenvolvimento, o projeto fornece representações do software que podem ser avaliadas quanto à qualidade. Existem diretrizes para a qualidade de um projeto de software e entre elas estão:

- ✖ **considerar e analisar todos os requisitos** especificados nos documentos de requisitos;
- ✖ **tornar-se um guia comprehensível** para aqueles que vão codificar, testar e manter o software;
- ✖ **prover um plano completo do software**, tratando aspectos funcionais, comportamentais e de dados, segundo uma perspectiva de implementação.

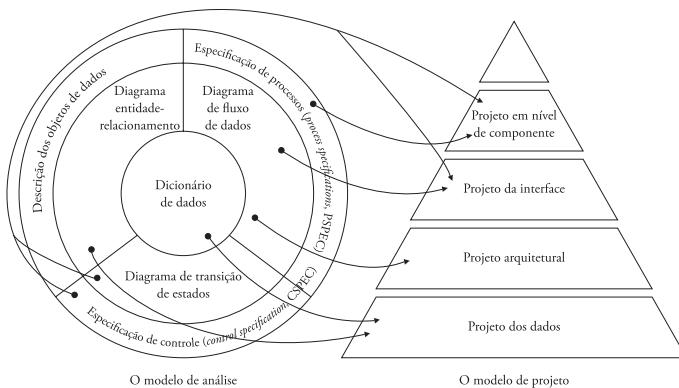
O projeto do software também serve de base para todo o processo de engenharia e manutenção de software que se seguirá. Sem o projeto, corre-se o risco de construir um sistema instável que poderá falhar quando pequenas mudanças forem feitas, que poderá ser difícil de se testar e cuja qualidade não poderá ser testada até um ponto tardio do processo de Engenharia de Software, quando o tempo é curto e muito dinheiro já foi investido.

Para Sommerville (2011), não há maneira certa ou errada de projetar software, aprende-se como projetar estudando exemplos de projetos existentes e discutindo o projeto com outros. A essência do projeto de software é tomar decisões sobre a organização lógica do software.

3.1.1 O modelo de análise e o modelo de projeto

Para entender o papel do projeto de software, é importante entender o fluxo de informações durante ele. Observe a Figura 3.1 e o texto explicativo na sequência.

Figura 3.1 – Tradução do modelo de análise em um modelo de projeto



Fonte: Pressman (2011).

Os elementos do modelo de análise fornecem informações para criar os quatro modelos de projeto necessários para uma completa especificação do projeto, tais como os diagramas pertencentes ao paradigma estruturado: dicionário de dados (DD), diagrama de transição de estado (DTE), diagrama de fluxo de dados (DFD), diagrama de entidade e relacionamento (DER), descrição dos objetos de dados, especificação de processos e de controle. Os requisitos de software, manifestados nos modelos de dados, funcional e comportamental, alimentam a tarefa de projeto, isto é, produzem um projeto de dados, um arquitetural, um da interface e um de componentes.

O projeto de dados transforma o modelo do domínio de informação, criado durante a análise, nas estruturas de dados que serão necessárias para implementar o software. Os objetos de dados e as relações definidas no DER, bem como o conteúdo detalhado dos dados mostrados no DD, fornecem a base para a atividade de projeto de dados. Já o projeto arquitetural define as relações entre os principais elementos estruturais do software, os “padrões de projeto”, que podem ser usados para satisfazer os requisitos que tenham sido definidos para o sistema e as restrições que afetam o modo pelo qual os padrões de projeto arquitetural podem ser aplicados. A representação desse tipo de projeto depende de um sistema baseado em computador – derivado da especificação do sistema, do modelo de análise e da interação dos subsistemas definida no modelo de análise.

O projeto da interface, por sua vez, descreve como o software se comunica com ele mesmo, com os sistemas que interagem com ele e com as pessoas que o utilizam. Uma interface implica em um fluxo de informação (dados e/ou controle) e um tipo de comportamento específico. O diagrama de fluxo de dados e controle fornece muita informação necessária para o projeto da interface. Por fim, o projeto de componentes transforma elementos estruturais da arquitetura de software em uma descrição procedural dos componentes de software.

3.1.2 Princípios do projeto de software

Alguns princípios para o projeto de software são sugeridos por Pressman (2011), tais como:

1. o projeto não pode ser “bitolado” – Um bom projetista deve considerar abordagens alternativas, julgando cada uma com base nos requisitos do projeto;
2. o projeto deve ser relacionável ao modelo de análise – Como um único elemento do projeto pode estar relacionado com vários requisitos, é necessário ter recursos para estabelecer como os requisitos serão satisfeitos pelo projeto;
3. o projeto não deve reinventar a roda – Os sistemas são construídos usando-se um conjunto de padrões de projeto, muitos dos quais estão descritos amplamente na literatura; os padrões de componentes devem ser escolhidos como alternativa à reinvenção (reuso de conhecimento);
4. o projeto deve “minimizar a distância intelectual” entre o software e o problema, tal como no mundo real – O software, sempre que possível, deve imitar a estrutura do domínio do problema;
5. o projeto deve exibir uniformidade e integração – Um projeto é uniforme se dá a ideia de que uma única pessoa o desenvolveu inteiramente. Regras de estilo e formato devem ser definidas para uma equipe de projeto antes que o trabalho seja iniciado; um projeto é integrado se houver o devido cuidado na definição de interfaces entre seus componentes;

6. o projeto deve ser estruturado para acomodar modificações;
7. o projeto deve ser estruturado para degradar suavemente, mesmo quando dados, eventos ou condições de operações aberrantes forem encontradas;
8. projeto não é codificação e codificação não é projeto – Mesmo quando os mais detalhados projetos procedimentais são criados para os componentes de programa, o nível de abstração de projeto é maior do que o do código-fonte;
9. o projeto deve ser avaliado quanto à qualidade à medida que é criado, não depois do fato – Uma variedade de conceitos e medidas de projeto está disponível para auxiliar o projetista na avaliação de qualidade;
10. o projeto deve ser revisto para minimizar erros conceituais (semânticos) – A equipe de projeto deve garantir que os principais elementos conceituais (omissões, ambiguidade e inconsistência) tenham sido cuidados antes de se preocupar com a sintaxe do modelo.

3.1.3 A qualidade do projeto de software

Conforme citado, o projeto de software é responsável por incorporar requisitos tecnológicos a requisitos essenciais (funcionais e não funcionais). Assim, o engenheiro de software deve estar atento aos critérios de qualidade que o sistema terá de atender. O modelo de qualidade definido na norma ISO/IEC 9126-1 (Organização Internacional para Padronização/Comissão Eletrotécnica Internacional), utilizado como referência para a avaliação de produtos de software, define seis características de qualidade, desdobradas em subcaracterísticas:

1. **funcionalidade** – refere-se à existência de um conjunto de funções que satisfazem às necessidades explícitas e as implícitas e suas propriedades específicas. Tem como subcaracterísticas: adequação, acurácia, interoperabilidade, segurança de acesso e conformidade;
2. **confiabilidade** – diz respeito à capacidade de o software manter seu nível de desempenho sob condições estabelecidas por um perí-

odo de tempo. Tem como subcaracterísticas: maturidade, tolerância a falhas, recuperabilidade e conformidade;

3. **usabilidade** – refere-se ao esforço necessário para se utilizar um produto de software, bem como o julgamento individual de tal uso por um conjunto de usuários. Tem como subcaracterísticas: inteligibilidade, apreensibilidade, operacionalidade, atratividade e conformidade;
4. **eficiência** – diz respeito ao relacionamento entre o nível de desempenho do software e a quantidade de recursos utilizados sob condições estabelecidas. Tem como subcaracterísticas: comportamento em relação ao tempo, comportamento em relação aos recursos e conformidade;
5. **manutenibilidade** – concerne ao esforço necessário para se fazer modificações no software. Tem como subcaracterísticas: analisabilidade, modificabilidade, estabilidade, testabilidade e conformidade;
6. **portabilidade** – refere-se à capacidade de o software ser transferido de um ambiente para outro. Tem como subcaracterísticas: adaptabilidade, capacidade para ser instalado, coexistência, capacidade para substituir e conformidade.

3.2 Os conceitos de projeto de software

Durante a elaboração do projeto de software é importante que alguns conceitos sejam levados em conta para que se possa derivar um projeto contendo as características citadas anteriormente. Na sequência, discutiremos alguns desses conceitos.

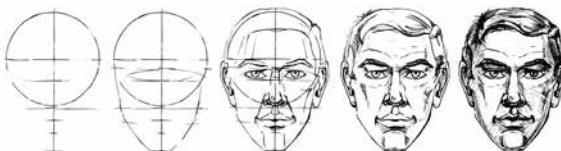
3.2.1 Abstração

O princípio de abstração está fortemente relacionado às características de modularidade que um software pode apresentar. Quando se desenvolve um software que apresentará essas características, é comum que suas representações sejam feitas considerando vários níveis de abstração.

No que diz respeito à linguagem de representação, nos níveis mais altos de abstração, a linguagem utilizada é bastante orientada à aplicação e ao ambiente no qual o software será executado. À medida que se “desce” nos níveis de abstração, a linguagem de representação se aproxima de questões de implementação até que, no nível mais baixo, a solução é representada de modo que possa ser derivada diretamente para uma linguagem de implementação. Na realidade, o próprio processo de Engenharia de Software constitui-se em um aprimoramento de níveis de abstração; por exemplo, parte das tarefas do sistema a desenvolver são atribuídas ao software, como elemento constituinte de um sistema computacional.

Observe que na vida real isso não é diferente. Veja a figura 3.2, que mostra a progressão da abstração do primeiro ao quinto estágio de uma ilustração, mostrando como o artista altera sua imagem à essência do tema.

Figura 3.2 – Abstração de ilustração



Fonte: Shutterstock.com/Babich Alexander.

3.2.2 Refinamento

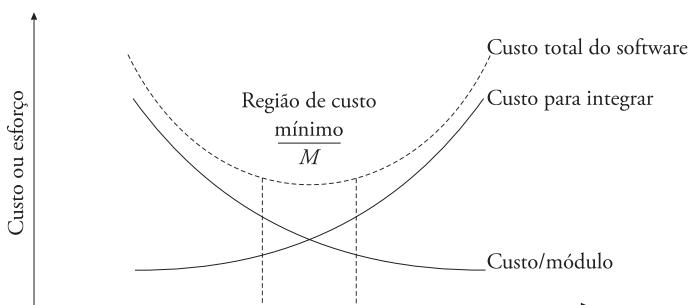
O refinamento surgiu como uma técnica de projeto que sugere como ponto de partida a definição da arquitetura do software a ser desenvolvido, sendo que esta é refinada sucessivamente até atingir níveis de detalhes procedimentais. Isso dá origem a uma hierarquia de representações na qual uma descrição macroscópica de cada função vai sendo decomposta passo a passo até se obter representações bastante próximas de uma linguagem de implementação. Esse processo é bastante similar ao utilizado em diversas técnicas de análise de requisitos (como o DFD e o SADT – Structured Analysis and Design Technique), sendo que a diferença fundamental está nos níveis de detalhamento atingidos nessa etapa.

3.2.3 Modularidade

O conceito de modularidade tem sido utilizado há bastante tempo como forma de obtenção de um software que apresente características interessantes como facilidade de manutenção. Apareceu como uma solução aos antigos softwares “monolíticos”, que representavam grandes dificuldades de entendimento e, consequentemente, dificuldades para qualquer atividade de manutenção. A utilização do conceito de modularidade oferece resultados de curto prazo, uma vez que, ao dividir-se um grande problema em problemas menores, as soluções são encontradas com esforço relativamente menor. Isso significa que quanto maior for o número de módulos definidos em um software, menor será o esforço necessário para desenvolvê-lo, uma vez que o esforço de desenvolvimento de cada módulo será menor. Por outro lado, quanto maior for o número de módulos, maior será o esforço no desenvolvimento das interfaces, o que permite concluir que essa regra deve ser utilizada com moderação.

É importante distinguir o conceito de modularidade de projeto do modelo de modularidade de implementação, mas nada impede que um software seja projetado sob a ótica da modularidade e que sua implementação seja monolítica. Em alguns casos, para evitar desperdício de tempo de processamento e de memória em chamadas de procedimentos (salvamento e recuperação de contexto), impõe-se o desenvolvimento de um programa sem a utilização de funções e de procedimentos. Ainda assim, uma vez que o projeto seguiu uma filosofia de modularidade, o software deverá apresentar alguns benefícios resultantes da adoção desse princípio.

Figura 3.3 – Modularidade e custo de software



Fonte: Pressman (2011).

3.2.4 A arquitetura de software

O conceito de arquitetura de software está ligado a dois principais aspectos do funcionamento de um software: a estrutura hierárquica de seus componentes (ou módulos) e as estruturas de dados. A arquitetura de software resulta do desenvolvimento de atividades de particionamento de problema, encaminhadas desde a etapa de análise de requisitos, na qual é dado o pontapé inicial para a definição das estruturas de dados e dos componentes de software.

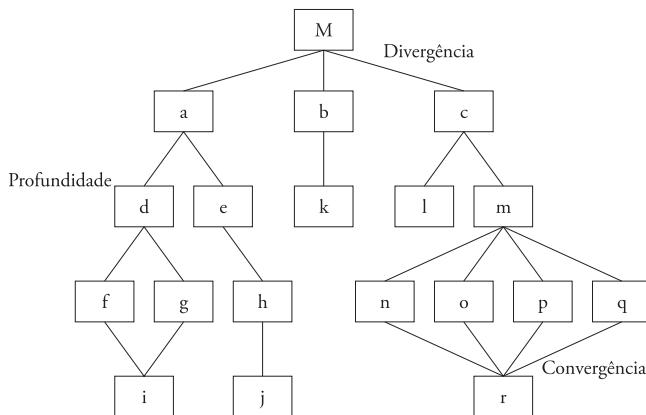
A solução é encaminhada ao longo do projeto, através da definição de um ou mais elementos de software que solucionarão uma parte do problema global. É importante lembrar que não existe técnica de projeto que garanta a unicidade de solução a um dado problema. Diferentes soluções em termos de arquitetura podem ser derivadas a partir de um mesmo conjunto de requisitos de software. A grande dificuldade concentra-se em definir qual é a melhor opção em termos de solução.

3.2.5 Hierarquia de controle

A hierarquia de controle nada mais é do que a representação da estrutura do software no que diz respeito a seus componentes. O objetivo não é apresentar detalhes procedimentais ou de sequenciamento entre processos, mas estabelecer as relações entre os diferentes componentes do software, explicitando os níveis de abstração (refinamento) aos quais eles pertencem.

O modo mais usual de apresentar a hierarquia de controle é em uma linguagem gráfica, normalmente em forma de árvore, como mostra a Figura 3.4. Com relação à estrutura de controle, é importante apresentar algumas definições de “medição”. Utilizando a Figura 3.4 como referência, é possível extrair alguns conceitos: a profundidade está associada ao número de níveis de abstração definidos para a representação do software; e a largura permite concluir sobre a abrangência do controle global do software.

Figura 3.4 – Árvore de hierarquia de controle



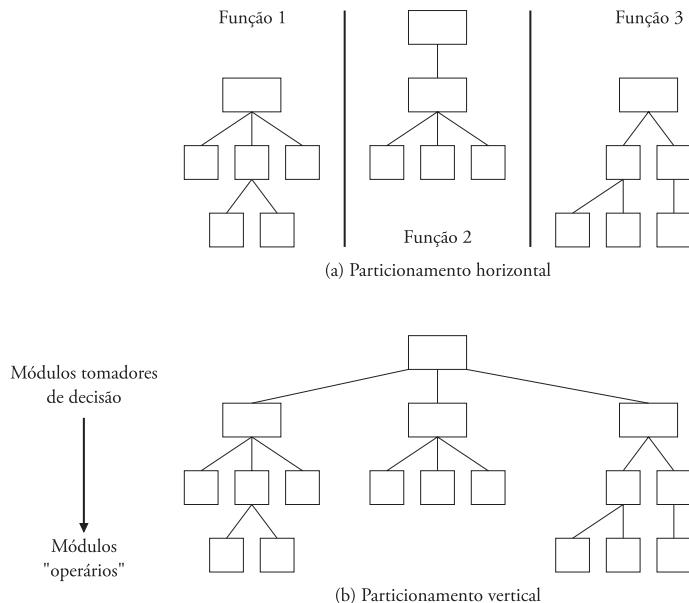
Fonte: Pressman (2011).

3.2.6 Estrutura dos dados

A estrutura dos dados representa os relacionamentos lógicos entre os diferentes elementos de dados individuais. À medida que o projeto se aproxima da implementação, essa representação assume fundamental importância, já que a estrutura da informação exerce impacto significativo no projeto procedural final. A estrutura dos dados define a forma como estão organizados os métodos de acesso, o grau de associatividade e as alternativas de processamento das informações.

Ainda que a forma de organizar os elementos de dados e a complexidade de cada estrutura dependam do tipo de aplicação a desenvolver e da criatividade do projetista, existe um número limitado de componentes clássicos que funcionam como blocos de construção de estruturas mais complexas. A Figura 3.5 apresenta alguns desses construtores, que podem ser combinados das mais diversas maneiras para obter estruturas de dados relativamente complexas.

Figura 3.5 – Particionamento estrutural



Fonte: Pressman (2011).

3.2.7 Procedimentos de software

Os procedimentos de software têm por objetivo expressar em detalhes e individualmente a operação de cada componente de software. Os aspectos a serem expressos nos procedimentos de software são o processamento da informação, o sequenciamento de eventos, os pontos de decisão, as operações repetitivas e, eventualmente (se houver necessidade), algumas estruturas de dados.

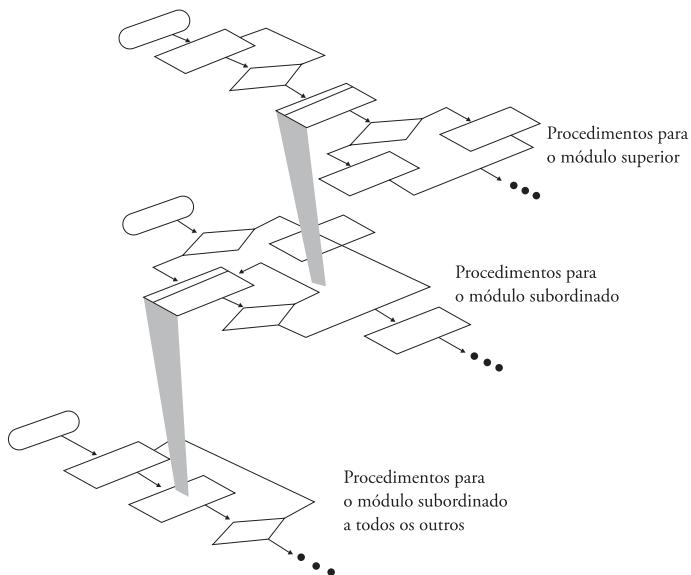
A construção de procedimentos de software deve ser realizada tendo como referência a hierarquia de controle porque, na definição do procedimento de um dado módulo de software, deve ser feita referência a todos os módulos subordinados a ele. Normalmente, essa referência é desenvolvida em um nível mais baixo de detalhamento, de modo a se obter o procedimento de software do módulo subordinado.

3.2.8 Ocultamento de informação

O princípio da ocultação de informações propõe um caminho para decompor sistematicamente um problema de modo a obter os diferentes módulos do software a se construir. Segundo esse princípio, os módulos devem ser decompostos para que as informações (os procedimentos e os dados) contidas em cada módulo sejam inacessíveis aos módulos que não tenham necessidade delas.

Ao realizar a decomposição segundo esse princípio, o engenheiro de software proporciona um grau relativamente alto de independência entre os módulos, o que é altamente desejável em tal projeto. Os benefícios da adoção desse princípio aparecem no momento em que modificações devem ser encaminhadas a nível da implementação, por exemplo, por consequência de uma atividade de teste do software. Devido à ocultação de informação, erros introduzidos como resultado de alguma modificação em um módulo não serão propagados a outros módulos do software.

Figura 3.6 – Procedimento em camadas



Fonte: Pressman (2011).

3.3 A modularização

Segundo Pressman (2011), os conceitos fundamentais de projeto trabalhados anteriormente servem para induzir projetos modulares. Um projeto modular reduz a complexidade, facilita a modificação (um aspecto crítico da manutenibilidade de software) e resulta em fácil implementação pelo incentivo ao desenvolvimento paralelo de diferentes partes de um sistema. A seguir, temos alguns conceitos da modularização, de acordo com Pressman (2011):

- × **independência funcional** – é uma decorrência direta da modularidade dos conceitos de abstração e de ocultamento funcional. A independência funcional é conseguida pelo desenvolvimento de módulos com função de “finalidade única” e “aversão” à interação excessiva com outros módulos. Ou seja, queremos projetar softwares de maneira que cada módulo cuide de uma subfunção específica dos requisitos e tenha uma interface simples quando visto de outras partes da estrutura do programa.

Módulos independentes são mais fáceis de manter e testar, porque os efeitos secundários causados por modificação de projeto ou código são limitados, a propagação de erros é reduzida e os módulos reusáveis são possíveis. Em suma, independência funcional é a chave para um bom projeto, e o projeto é a chave da qualidade de software. Independência, por sua vez, é medida usando dois critérios qualitativos:

- × **coesão** – um módulo coeso realiza uma única tarefa dentro de um procedimento de software, requerendo pouca interação com procedimentos que estão sendo realizados em outras partes de um programa. Assim, um módulo coeso deveria idealmente fazer apenas uma coisa;
 - Alta coesão → Excelente
 - Baixa coesão → Problemas
- × **acoplamento** – medida da interconexão entre módulos em uma estrutura de software. O acoplamento depende da complexidade da interface entre módulos, do ponto em que é feita a entrada ou referência a um módulo e que dados passam pela

interface. Em projeto de software, buscamos o acoplamento mais baixo possível. Conectividade simples entre módulos resulta em software mais fácil de ser entendido e menos propenso a “efeito de propagação”, que acontece quando erros que ocorrem em um lugar se propagam por todo o sistema.

3.3.1 A heurísticas de projeto para modularidade

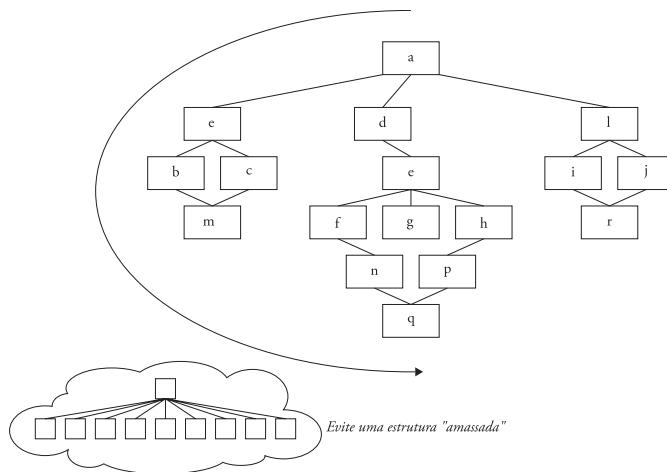
Primeiramente, devemos entender o termo *heurística*: método não comprovado cientificamente, ou seja, que não tem confirmação matemática. Portanto, uma decisão tomada por experiência na função, na intuição, no bom senso ou em outra forma que não seja confirmada por um método matemático trata-se de confirmação heurística.

Segundo Pressman (2011), a estrutura do programa pode ser manipulada de acordo com o seguinte conjunto de heurísticas:

- ✖ avalie a “primeira iteração” da estrutura do programa para reduzir o acoplamento e melhorar a coesão – uma vez desenvolvida a estrutura do programa, os módulos podem ser explodidos ou implodidos com o objetivo de melhorar a independência modular. Um módulo explodido transforma-se em dois ou mais módulos na estrutura final do programa. Um módulo implodido é o resultado da combinação do processamento implícito em dois ou mais módulos.
- ✖ avalie as interfaces do módulo para reduzir a complexidade e a redundância e aperfeiçoar a consistência – a complexidade de interface de um módulo é uma causa importante de erros de software. Interfaces devem ser projetadas para passar informação de modo simples e devem ser consistentes com a função do módulo.
- ✖ defina módulos cuja função seja previsível, mas evite módulos que sejam excessivamente restritivos – um módulo que restringe o processamento a uma única subfunção exibe alta coesão e é visto favoravelmente por um projetista.
- ✖ procure obter módulos “de entrada controlada” evitando “conexões patológicas” – um software é mais fácil de ser entendido e consequentemente mais fácil de ser mantido quando as interfaces de

módulos são restritas e controladas. Conexão patológica refere-se a desvios ou referências no meio de um módulo.

Figura 3.7 – Estrutura de programa



Fonte: Pressman (2011).

3.3.2 Métodos para modularização

Na sequência, são apresentados diferentes métodos propostos na literatura para ajudar as organizações a desenvolver sistemas modulares.

3.3.2.1 Design Structure Matrix

Segundo Thebeau (2001), o Design Structure Matrix (DSM) é uma ferramenta útil para representar as relações entre elementos diferentes, que podem ser componentes físicos, sistemas, parâmetros ou qualquer outro item no qual possa ocorrer interação e interface. O DSM é uma matriz na qual os componentes do produto são registrados nas linhas e nas colunas; para cada cruzamento de linha e coluna, a equipe assinala se um grupo de componentes é fisicamente conectado a outro e assume-se que é melhor que fiquem em um mesmo módulo do que separados. O sistema é construído, basicamente, por três passos: decomposição do sistema em elementos; análise das interações físicas e funcionais entre os elementos; e identificação dos potenciais agrupamentos.

A decomposição do sistema em elementos consiste na descrição do conceito do produto em termos de elementos físicos e funcionais que permitam atingir as funções do produto. A análise das interações físicas e funcionais entre os elementos visa documentar as interações entre os elementos, que podem ser espaciais, de energia, de informação ou de material. A identificação dos potenciais agrupamentos objetiva agrupar os elementos em blocos com base em critérios pré-estabelecidos pela equipe. Esses blocos definem a arquitetura do produto.

3.3.2.2 Design for Variety

Segundo Simpson et al. (2012), em 2002 Mark Martin e Kosuke Ishii propuseram um método para incorporar padronização e modularização com o objetivo de reduzir futuros custos e esforços de projeto. O método é composto por quatro etapas e se inicia com a geração do Generational Variety Index (GVI) e do Coupling Index (CI). GVI é um indicador da quantidade esperada de redesenho necessário para que um componente satisfaça as exigências do mercado no futuro, valor atribuído pela equipe e baseado em fatores como as necessidades dos consumidores, a confiabilidade de componentes e os custos. CI é um indicador da probabilidade de a alteração em um componente exigir redesenho de outros componentes.

Em seguida, os componentes são ordenados do maior para o menor conforme seu GVI e com o CI são determinados os componentes com maiores chances de serem trocados e como essa troca se propagará no restante da estrutura. Com o GVI e o CI, a equipe de projeto pode começar a fazer modificações na arquitetura dos produtos e desenvolver uma plataforma de produto mais facilmente aplicada às gerações futuras do produto. No terceiro passo, é necessário decidir quais partes do produto serão modularizadas e quais serão padronizadas.

A modularização tem relação direta com o CI, pois, quando ele é zero, os componentes podem ser trocados para atender às expectativas dos clientes sem a necessidade de que outros componentes também sejam trocados. Por fim, é aplicada uma abordagem prescritiva para aperfeiçoar a arquitetura do produto, ajudando nas decisões de como arranjar os componentes e como definir interfaces.

3.3.2.3 Fuzzy Logic Based

O método proposto por Nepal, Monplaisir e Singh (2005), utiliza-se de método baseado na lógica difusa (Fuzzy) para lidar com o conhecimento vago e impreciso nos estágios iniciais de desenvolvimento de produto. Seu objetivo é otimizar a arquitetura do produto com vista aos custos, mas também com relação a qualidade, confiabilidade e manufaturabilidade. O método proposto inicia com a aquisição do conhecimento, com o qual é realizada análise do produto, identificação das variáveis linguísticas e desenvolvimento de regras. Então, forma-se um número de potenciais módulos, utilizando-se as variáveis linguísticas para representar o custo desses módulos.

A principal função das variáveis linguísticas é fornecer uma maneira sistemática para caracterização aproximada de fenômenos complexos ou mal definidos. Nesse caso, o conhecimento vago e impreciso característico das fases iniciais de desenvolvimento. Na sequência, ocorre um processo de inferência difusa e o processo de modularização do produto é feito por meio de um modelo matemático baseado em um algoritmo de tecnologia de grupo, visando à minimização do custo de modularização de uma arquitetura de produto.

3.3.2.4 House of Modular Enhancement

A metodologia Home (House of Modular Enhancement) visa adicionar questões relacionadas ao ciclo de vida do produto ainda na fase de projeto, por meio do redesign de produtos existentes para o aprimoramento da modularidade (SAND; GU; WATSON, 2002). Para isso, deve-se estabelecer relações entre os objetivos e os módulos.

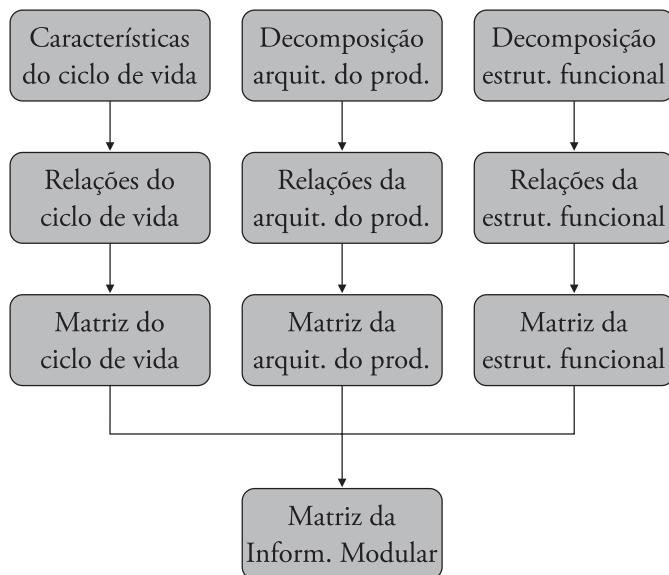
Uma matriz de informação do design modular é gerada, incluindo três tipos de informações principais: objetivos do ciclo de vida (manutenção, reuso, reciclagem), informação de arquitetura do produto e requerimentos funcionais, gerando nove matrizes. Essas matrizes são combinadas para formar a Modular Information Matrix (MIM), que provê informações das relações diretas entre os componentes.

A matriz MIM é transformada por meio da utilização de método de eixo radial (Radial Axis Method – RAM), gerando a matriz EMIM, na qual um algoritmo de agrupamento é aplicado para integrar os componentes em módulos do produto. Segundo os autores, a metodologia identifica fatores

relacionados aos objetivos, relaciona esses fatores aos componentes através de análise de interações e os agrupa em módulos.

A matriz do ciclo de vida relaciona os componentes com componentes com relação a aspectos do ciclo de vida, como reuso e reciclagem. A matriz da arquitetura do produto analisa as relações físicas entre os componentes dos produtos, e a matriz da estrutura funcional apresenta informações a respeito das relações funcionais entre os componentes. A matriz MIM é uma combinação dessas três matrizes.

Figura 3.8 – Organização das matrizes



Fonte: Sand; Gu; Watson (2002).

3.4 Principais métodos para o projeto de software

Como um complemento, na sequência são apresentados alguns dos principais métodos para ajudar a orientar o processo de planejamento do projeto de software.

- × **Projeto orientado por função (estruturado):** um dos métodos clássicos do projeto de software, no qual a decomposição centra na identificação das principais funções de software e refinação, elaborando-as de cima para baixo. O projeto estruturado é geralmente usado depois da análise estruturada, assim, produção, entre outras coisas, diagrama de fluxos de dados e descrições de processo associadas.
- × **Projeto orientado a objeto (OO):** numerosos métodos de design de software baseados em objetos já foram propostos. O campo evoluiu com o primeiro design baseado em objeto em meados dos anos 1980 (substantivo = objeto; verbo = método; adjetivo = atributo) pelo design de OO, com o qual a herança e o polimorfismo desempenham um papel-chave, do campo do design à base de componente, onde a informação de Meta pode ser definida e acessada (pela reflexão, por exemplo). Embora as raízes de design OO venham do conceito de abstração de dados, o design levado por responsabilidade também teria sido proposto como uma aproximação alternativa ao design de OO.
- × **Projeto centrado por estruturas de dados:** parte da estrutura de dados que um programa manipula, e não da função que ele executa. O engenheiro de software primeiro descreve os dados de produção e a entrada das estruturas e logo desenvolve a estrutura de controle do programa baseada nos diagramas de estrutura de dados. Várias heurísticas foram propostas para tratar com casos especiais, por exemplo, quando há uma má combinação entre as estruturas de produção e entrada.
- × **Projeto baseado em componentes:** um componente de software é uma unidade independente, tendo interfaces bem definidas e dependências que podem ser compostas e desdobradas separadamente. O projeto baseado em componentes dirige questões relacionadas a fornecimento, desenvolvimento e integração de tais componentes para melhor reutilização.

Síntese

Vimos neste capítulo que o projeto de software é a fase de desenvolvimento na qual são feitos modelos com todas as entidades que serão construídas posteriormente a partir dos requisitos do sistema. O projeto de software trabalha com quatro níveis de detalhamento: dados, arquitetura, interface e componentes.

Também verificamos que, para garantir que um projeto está sendo realizado com qualidade, o engenheiro de software deve estar atento aos critérios de qualidade que o sistema terá de atender, tais como funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade e portabilidade.

Atividades

1. Em sua opinião, por que o projeto de software é importante?
2. Descreva os principais objetivos dos quatro níveis de detalhes de um projeto de software.
3. O que é confiabilidade para a qualidade do projeto de software?
4. Descreva o conceito de coesão.

4

Construção de Software

O TERMO CONSTRUÇÃO de software refere-se à criação detalhada de software de trabalho por meio de uma combinação de codificação, verificação, teste unitário, testes de integração e depuração.

ESSA ETAPA USA a saída do projeto e fornece uma entrada para o teste. Os limites entre o projeto, a construção e os testes podem variar de acordo com modelo de ciclo de vida do software usado.

O OBJETIVO DESTE capítulo é apresentar a fundamentação da construção de software, além de discutir o gerenciamento da construção de software e suas práticas.

4.1 Fundamentação da construção de software

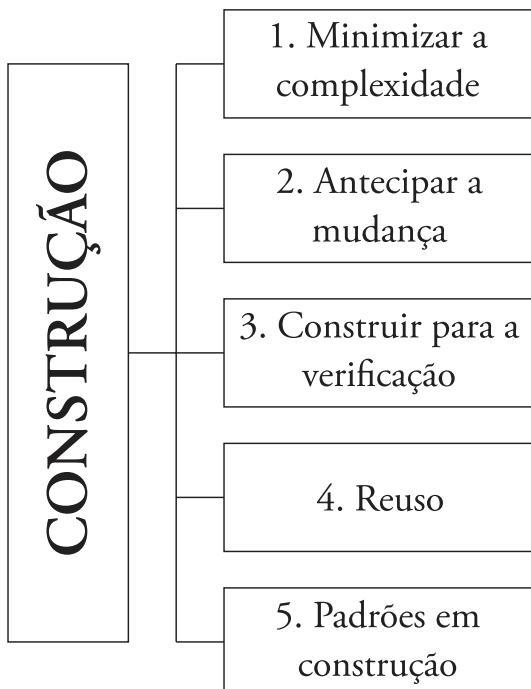
A CONSTRUÇÃO DE software está relacionada a implementação do software, verificação, teste unitário, teste de integração e testes de sistema. Pode-se afirmar que a construção de software

está envolvida com todas as áreas de conhecimento da engenharia de software, entretanto existe um relacionamento maior com o projeto de software e com teste de software.

Após o software ser projetado, é necessário escrever e testar os códigos que implementam tal projeto. No entanto, não se engane. Por mais que o projeto seja bem elaborado, a implementação do software não é uma tarefa fácil. Por vezes, os projetistas do software não conhecem profundamente a plataforma de implementação, o que pode deixar lacunas no projeto e prejudicar a construção do software.

O SWEBOK (IEEE, 2004), apresenta cinco pilares para auxiliar a fase da construção do software. Observe a figura e, em seguida, a descrição de cada um desses pilares.

Figura 4.1 – Construção de software



Fonte: Adaptado de IEEE (2004).

1. Procure **minimizar a complexidade**: a maioria das pessoas tem dificuldade de lidar e manter estruturas e informações complexas no trabalho, especialmente durante longos períodos de tempo. Por esse motivo, é importante procurar sempre reduzir a complexidade durante a construção do software, ou seja, criar códigos simples, bem definidos e legíveis aos demais participantes do projeto. Para auxiliar esse direcionamento, existem diversas técnicas, como normas de codificação e qualidade da construção.
2. Busque **anticipar a mudança**: os softwares sofrem mudanças ao longo do tempo, e a antecipação da mudança impulsiona vários aspectos da construção do software. As mudanças nos ambientes nos quais o software opera também afetam o software de diversas maneiras. A antecipação a mudanças ajuda os engenheiros de software a construir um software extensível, o que significa que eles podem aprimorar um produto de software sem interromper a estrutura subjacente.
3. Pense em **construir para a verificação**: isso significa construir o software de tal forma que as falhas possam ser prontamente encontradas pelos engenheiros de software que o escrevem, bem como pelos testadores e usuários durante os testes. As técnicas específicas que suportam a construção para verificação incluem: padrões de codificação para suportar revisões de código e teste unitário, organização de código para suportar testes automatizados e restrição do uso de estruturas de linguagem complexas ou difíceis de entender.
4. Utilize o **reuso**: refere-se ao uso de elementos existentes na solução de problemas diferentes. Na construção de software, os elementos tipicamente reutilizados incluem bibliotecas, módulos, componentes e código-fonte. A reutilização é mais bem praticada sistematicamente, de acordo com um processo bem definido e repetitivo. A reutilização sistemática pode permitir produtividade significativa, qualidade e melhorias de custo do software. Observe que a reutilização tem dois propósitos estreitamente relacionados: (1) construção para reutilização e (2) construção com reutilização. O primeiro significa criar elementos de software reutilizáveis, enquanto o segundo significa reutilizar ativos de software na construção de

uma nova solução. A reutilização, muitas vezes, transcende a fronteira dos projetos, o que significa que os elementos reutilizados podem ser construídos em outros projetos ou organizações.

5. Aplique **padrões em construção**: a aplicação de padrões de desenvolvimento externos ou internos durante a construção ajuda a alcançar os objetivos de um projeto em termos de eficiência, qualidade e custo. Especificamente, as escolhas de subconjuntos de linguagem de programação e padrões de uso são auxílios importantes para alcançar maior segurança. Os padrões que afetam diretamente as questões de construção incluem:
 - i. **métodos de comunicação** (por exemplo, padrões para formatos e conteúdo de documentos);
 - ii. **linguagens de programação** (por exemplo, padrões de linguagem de programação como Java e C ++);
 - iii. **padrões de codificação** (por exemplo, padrões para convenções de nomenclatura, leiaute e recuo);
 - iv. **plataformas** (por exemplo, padrões de interface para chamadas do sistema operacional);
 - v. **ferramentas** (por exemplo, padrões diagramáticos para notações como UML (*Unified Modeling Língua*)).

4.2 Gerenciamento da construção de software

Como vimos nos capítulos anteriores, diversos modelos foram criados para desenvolver softwares. No entanto, alguns enfatizam a construção mais do que outros.

Os modelos sequenciais são mais lineares do ponto de vista da construção, como os modelos em cascata e em V. Estes tratam a construção como uma atividade que ocorre somente após o trabalho de levantamento/detalhamento dos requisitos e da elaboração do projeto detalhado. Dessa maneira, as abordagens mais lineares tendem a enfatizar as atividades que precedem a construção e a criar separações mais distintas entre as atividades. Nesses modelos, a ênfase principal da construção pode ser a codificação.

Já os modelos mais iterativos, como prototipação e desenvolvimento ágil, tendem a tratar a construção como uma atividade que ocorre simultaneamente com outras atividades de desenvolvimento de software ou que as sobrepõe. Essas abordagens tendem a misturar atividades de projeto, codificação e teste, e muitas vezes tratam da combinação de atividades como “construção”.

Consequentemente, o que é considerado “construção” depende, em certa medida, do modelo de ciclo de vida utilizado. Em geral, a construção de software é principalmente codificação e depuração, mas também envolve planejamento de construção, projeto detalhado, testes unitários, testes de integração e outras atividades.

De acordo com McConnell (2004), no planejamento a escolha do método de construção é um aspecto-chave, visto que afeta a extensão em que os requisitos de construção são executados, a ordem em que eles são realizados e o grau em que eles devem estar concluídos antes do início dos trabalhos de construção. Tal abordagem da construção afeta a capacidade da equipe do projeto de reduzir a complexidade, antecipar a mudança e construir para a verificação.

O planejamento de construção também define a ordem na qual os componentes são criados e integrados, a estratégia de integração (por exemplo, integração gradual ou incremental), os processos de gerenciamento de qualidade do software e a alocação das tarefas.

Como um mecanismo para garantir a qualidade durante a construção e melhorar o processo de construção, algumas atividades e artefatos podem ser medidos, como: código-fonte, complexidade de código, estatísticas de inspeção de código, falhas e taxas de busca de falhas, esforço e agendamento.

4.3 Considerações práticas da construção de software

A construção é uma atividade em que o engenheiro de software tem que lidar com restrições do mundo real – às vezes caóticas e mutáveis. Mediante a forte influência de restrições do mundo real, a construção pode ser impulsionada por considerações práticas. A seguir, vamos observar algumas dessas

considerações apresentadas por Sommerville (2011) e também disponíveis no SWEBOK (IEEE, 2004).

1. **Projeto de construção:** algumas empresas atribuem uma atividade de elaboração do projeto na construção, enquanto outras alocam a elaboração do projeto de construção a uma fase explicitamente focada no projeto de software. Independentemente da alocação exata, algum detalhamento do projeto ocorre no nível da construção, visto que, por vezes, o projeto de construção tende a ser ditado pelas limitações impostas pelo problema do mundo real que está sendo resolvido pelo software.
2. **Linguagem de construção:** incluem todas as formas de comunicação pelas quais uma pessoa pode especificar uma solução de um problema. As linguagens de construção e suas implementações (por exemplo, compiladores) podem afetar os atributos de qualidade de software de desempenho, confiabilidade, portabilidade e assim por diante.

O tipo mais simples de linguagem de construção é uma linguagem de configuração, na qual os engenheiros de software escolhem a linguagem dentre um conjunto limitado de opções predefinidas para criar instalações de software novas ou personalizadas. Os arquivos de configuração baseados em texto, usados nos sistemas operacionais *Windows* e *Unix*, são exemplos disso.

Já as linguagens de *toolkit* são usadas para construir aplicações fora de elementos em *toolkits* (conjuntos integrados de partes reutilizáveis específicas de aplicação), que são mais complexos do que as linguagens de configurações. As linguagens do *toolkit* podem ser explicitamente definidas como linguagens de programação de aplicativo.

As linguagens de *script* são tipos comumente usados de linguagens de programação de aplicativos. Em algumas linguagens de *script*, os *scripts* são chamados de arquivos em lote ou macros.

Já as linguagens de programação são o tipo mais flexível de linguagens de construção, contendo a menor quantidade de informações sobre áreas de aplicação específicas e processos de desenvolvimento;

portanto, exigem o máximo de treinamento e habilidade para serem usadas de maneira eficaz. A escolha da linguagem de programação pode ter um grande efeito na probabilidade de introdução de vulnerabilidades durante a codificação – por exemplo, o uso de C e C ++ são escolhas questionáveis do ponto de vista da segurança. Existem três tipos gerais de notação usados para linguagens de programação, a saber:

- ✗ linguística (por exemplo, C / C ++, Java);
- ✗ formal (por exemplo, Event-B);
- ✗ visual (por exemplo, MatLab).

As notações linguísticas distinguem-se, em particular, pelo uso de cadeias textuais para representar construções de software complexas. A combinação de cadeias textuais em padrões pode ter uma sintaxe como uma frase. Adequadamente utilizados, cada *string* deve ter uma forte conotação semântica, fornecendo uma compreensão intuitiva imediata do que acontecerá quando a construção do software for executada.

As notações formais dependem menos do significado intuitivo e cotidiano das palavras e mais de definições apoiadas em definições precisas, inequívocas e formais (ou matemáticas). As construções formais usam modos precisamente definidos de combinar símbolos que evitam a ambiguidade de muitas construções de linguagem natural.

As notações visuais dependem muito menos das notações textuais da construção linguística e formal e, em vez disso, dependem da interpretação visual direta e da colocação de entidades visuais que representam o software subjacente. A construção visual tende a ser um pouco limitada pela dificuldade de fazer afirmações “complexas” usando apenas a disposição de ícones em uma tela. No entanto, esses ícones podem ser ferramentas poderosas nos casos em que a tarefa de programação primária é simplesmente construir e ajustar uma interface visual para um programa, cujo comportamento detalhado tem uma definição subjacente.

3. Codificação: as considerações a seguir se aplicam à atividade de codificação de construção de software.

- ✗ Técnicas para criar código-fonte comprehensível, incluindo convenções de nomenclatura e leiaute de código-fonte.
- ✗ Uso de classes, tipos *enumeration*, variáveis, *constraints* nomeadas e outras entidades similares.
- ✗ Uso de estruturas de controle.
- ✗ Manipulação de condições de erro, tanto antecipadas como excepcionais (entrada de dados incorretos, por exemplo).
- ✗ Prevenção de falhas de segurança de nível de código (*buffer overflows*, por exemplo).
- ✗ Uso de recursos por mecanismos de exclusão e acesso a recursos reutilizáveis.
- ✗ Organização de código-fonte (em rotinas, classes, pacotes ou outras estruturas).
- ✗ Documentação de código.

4. Teste de construção: a construção envolve duas formas de teste, que são muitas vezes realizadas pelo engenheiro de software que escreveu o código: (1) teste unitário e (2) teste de integração. O objetivo dos testes de construção é diminuir o intervalo entre o momento em que as falhas são inseridas no código e o momento em que essas falhas são detectadas, reduzindo assim o custo da correção. Em alguns casos, os casos de teste são escritos depois que o código foi construído. Em outros casos, os casos de teste podem ser criados antes do código ser construído.

5. Construção para reutilização: a construção para reutilização procura criar um software com o potencial de ser reutilizado no futuro para o presente projeto ou outros projetos. A construção para reutilização é geralmente baseada na análise de variabilidade. Para evitar o problema da duplicação de código, é desejável encapsular fragmentos de código reutilizáveis em bibliotecas ou componentes bem

estruturados. As tarefas relacionadas à construção de software para reutilização durante a codificação e teste são as seguintes:

- ✗ implementação da variabilidade com mecanismos como parametrização, compilação condicional, padrões de projeto, entre outros;
- ✗ encapsulamento de variabilidade para tornar os recursos de software fáceis de configurar e personalizar;
- ✗ testar a variabilidade fornecida pelos ativos de software reutilizáveis;
- ✗ descrição e publicação de ativos de software reutilizáveis.

6. Construção com reutilização: construção com reutilização significa criar um novo software com a reutilização de ativos de software existentes. O método mais popular de reutilização é reutilizar código das bibliotecas fornecidas por linguagem, plataforma, ferramentas utilizadas ou repositório organizacional. Além destas, as aplicações desenvolvidas atualmente fazem uso de muitas bibliotecas de código aberto. O software reutilizado e de prateleira muitas vezes têm os mesmos requisitos de qualidade, ou são até mesmo melhores. As tarefas relacionadas à construção de software com reutilização durante codificação e teste são as seguintes:

- ✗ seleção das unidades reutilizáveis, bancos de dados, procedimentos de teste ou dados de teste;
- ✗ avaliação de código ou teste de reutilização;
- ✗ integração de ativos de software reutilizáveis no software atual;
- ✗ relato de informações de reutilização sobre novos códigos, procedimentos de teste ou dados de teste.

7. Qualidade da construção: além de falhas resultantes de requisitos e projeto, falhas introduzidas durante a construção podem resultar em sérios problemas de qualidade, por exemplo vulnerabilidades de segurança. Isso inclui não apenas falhas na funcionalidade de segurança, mas também falhas em outros lugares que permitem ignorar

essa funcionalidade e outras fraquezas ou violações de segurança. Existem inúmeras técnicas para garantir a qualidade do código à medida que ele é construído. As principais técnicas utilizadas para a qualidade da construção incluem:

- ✗ teste de unidade e teste de integração;
- ✗ uso de programação defensiva;
- ✗ depuração;
- ✗ inspeções;
- ✗ revisões técnicas, incluindo avaliações orientadas para a segurança;
- ✗ análise estática.

A técnica ou as técnicas específicas selecionadas dependem da natureza do software a ser construído, bem como da competência dos engenheiros de software que executam as atividades de construção. Os programadores devem conhecer boas práticas e vulnerabilidades comuns – por exemplo, a análise estática automatizada do código para falhas de segurança, que está disponível para várias linguagens de programação e pode ser usada em projetos críticos de segurança.

As atividades de qualidade da construção são diferenciadas de outras atividades de qualidade pelo seu foco. Elas se concentram em código e artefatos intimamente relacionados ao código, ao contrário de outros artefatos que não estão tão conectados ao código, como requisitos e projetos de alto nível.

8. Integração: uma atividade-chave durante a construção é a integração de rotinas, classes, componentes e subsistemas individualmente construídos em um único sistema.

Além disso, determinado sistema de software pode precisar ser integrado a outros softwares ou sistemas de hardware. As preocupações relacionadas com a integração da construção incluem o planejamento da sequência em que os componentes serão integrados, a identificação do hardware necessário, a criação de andaimes para

suportar versões provisórias do software, a determinação do grau de teste e qualidade do trabalho realizado antes de serem integrados.

Os programas podem ser integrados por meio da abordagem gradual ou incremental. A integração faseada, também chamada integração de “*big bang*”, implica atrasar a integração de componentes de software até que todas as peças destinadas a serem lançadas em uma versão estejam completas. A integração incremental oferece muitas vantagens sobre a tradicional integração faseada, como a mais fácil localização de erros, melhorado monitoramento de progresso, entrega de produtos mais adiantada e melhores relações com o cliente.

Na integração incremental, os desenvolvedores escrevem e testam um programa em pequenos pedaços e, em seguida, combinam as peças de cada vez. Infraestrutura de teste adicional, como *stubs*, drivers e objetos simulados, geralmente são necessários para permitir a integração incremental. Ao construir e integrar uma unidade de cada vez (por exemplo, uma classe ou componente), o processo de construção pode fornecer *feedback* imediato aos desenvolvedores e clientes.

4.4 Tecnologias da construção de software

- 1. Projeto e uso da API:** uma interface de programação de aplicativo, mais conhecida como API (*Application Programming Interface*), é o conjunto de assinaturas que são exportadas e disponibilizadas para os usuários de uma biblioteca ou uma estrutura para gravar seus aplicativos. Além das assinaturas, uma API deve sempre incluir declarações sobre os efeitos e/ou comportamentos do programa. O *design* da API deve tentar torná-la fácil de aprender e memorizar, levar o código legível, ser fácil de estender, ser completo e manter compatibilidade com versões anteriores. Como as APIs geralmente ultrapassam suas implementações para uma biblioteca ou framework amplamente utilizado, é desejável que seja direta e mantenha-se estável para facilitar o desenvolvimento e a manutenção dos aplicativos-cliente.

2. **Problemas de tempo de execução:** linguagens orientadas a objetos suportam uma série de mecanismos de tempo de execução, incluindo polimorfismo e reflexão. Tais mecanismos aumentam a flexibilidade e a adaptabilidade dos programas orientados a objetos. Como o programa não conhece os tipos exatos dos objetos com antecedência, o comportamento exato é determinado em tempo de execução (chamado de vinculação dinâmica). Reflexão é a capacidade de um programa de observar e modificar sua própria estrutura e comportamento em tempo de execução. A reflexão permite a inspeção de classes, interfaces, campos e métodos em tempo de execução sem saber seus nomes em tempo de compilação. Ele também permite a instanciação em tempo de execução de novos objetos e a invocação de métodos usando nomes de classes e métodos parametrizados.
3. **Parametrização:** os tipos parametrizados, também conhecidos como genéricos e *templates*, permitem a definição de um tipo ou classe sem especificar todos os outros tipos que ele usa. Os tipos não especificados são fornecidos como parâmetros no ponto de utilização; os tipos parametrizados fornecem uma terceira via, além da herança de classe e composição de objeto, para compor comportamentos no software orientado a objeto.
4. **Asserções e programação defensiva:** uma asserção é um processo executável colocado em um programa (normalmente uma rotina ou macro), que permite verificações de tempo de execução do programa. As asserções são especialmente úteis em programas de alta confiabilidade, pois permitem aos programadores encontrar as interfaces incompatíveis, os erros que ocorrem quando o código é modificado e assim por diante.

Já a programação defensiva significa proteger uma rotina de ser quebrada por entradas inválidas. Maneiras comuns de lidar com entradas inválidas incluem verificar os valores de todos os parâmetros de entrada e decidir como lidar com entradas ruins. As asserções são frequentemente usadas na programação defensiva para verificar os valores de entrada.

5. Manipulação de erros, exceções e tolerância a falhas: a maneira como os erros são manipulados afeta a capacidade do software de atender aos requisitos relacionados a correção, robustez e outros atributos não funcionais. As asserções são algumas vezes usadas para verificar erros. Outras técnicas de tratamento de erros, como retornar um valor nulo, substituir o próximo pedaço de dados válidos, registrar uma mensagem de aviso ou retornar um código de erro, também são usados.

As exceções são usadas para detectar e processar erros ou eventos excepcionais. A estrutura básica de uma exceção é que uma rotina usa *throw* para lançar uma exceção detectada e um bloco de tratamento de exceções irá pegar a exceção em um bloco *try-catch*. O bloco *try-catch* pode processar a condição errada na rotina ou pode retornar o controle para a rotina chamadora. As políticas de tratamento de exceções devem ser cuidadosamente projetadas seguindo princípios comuns, como incluir na mensagem de exceção todas as informações que levaram à exceção, evitar blocos de captura vazios, conhecer as exceções que o código da biblioteca lança, talvez construir um retorno de exceção centralizado e padronizar o uso do programa.

Tolerância a falhas é uma coleção de técnicas que aumentam a confiabilidade do software, detectando erros e, em seguida, recuperando-se deles se possível ou contendo seus efeitos, se a recuperação não é possível. As estratégias de tolerância a falhas mais comuns incluem: fazer backup, usar código auxiliar e substituir um valor incorreto por um valor falso que terá um efeito benigno.

6. Modelos executáveis: os modelos executáveis abstraem os detalhes de linguagens de programação específicas e as decisões sobre a organização do software. Diferentemente dos modelos de software tradicionais, uma especificação construída em uma linguagem de modelagem executável como xUML (UML executável) pode ser implementada em vários ambientes de software sem alterações. Um compilador de modelo executável pode transformar um modelo em uma implementação usando um conjunto de definições sobre

o ambiente de hardware e software de destino. Assim, os modelos executáveis podem ser considerados como uma forma de construir software executável. Um modelo executável é uma maneira de especificar completamente um Modelo Independente de Plataforma (PIM), que é um modelo de solução para um problema que não depende de qualquer tecnologia de implementação. Dessa maneira, um Modelo Específico de Plataforma (PSM), que é um modelo que contém os detalhes da implementação, pode ser produzido.

7. **Técnicas de construção baseadas em estados:** programação baseada em estados, ou programação baseada em autômatos, é uma tecnologia de programação que usa máquinas de estado para descrever comportamentos de programa. Os gráficos de transição de uma máquina de estado são usados em todos os estágios de desenvolvimento de software (especificação, implementação, depuração e documentação). A ideia principal é construir programas de computador da mesma forma que a automação dos processos tecnológicos é feita. A programação baseada em estado é geralmente combinada com a programação orientada a objetos, formando uma nova abordagem composta chamada programação orientada a objetos baseada no estado.

Um método baseado em tabelas é um esquema que usa tabelas para procurar informações em vez de usar instruções de lógica. Usado em circunstâncias apropriadas, o código baseado em tabelas é mais simples e mais fácil de modificar. Ao usar métodos baseados em tabelas, o programador aborda dois problemas: quais informações armazenar na tabela ou tabelas e como acessar eficientemente as informações.

8. **Configuração de tempo de execução e internacionalização:** a configuração de tempo de execução é uma técnica que vincula os valores das variáveis e as configurações do programa quando este está em execução, normalmente atualizando e lendo os arquivos de configuração em um modo *just-in-time*. A internacionalização é a atividade técnica de preparar um programa, geralmente um software interativo, para suportar vários idiomas. A atividade corres-

pondente – a localização – é a atividade de modificar um programa para suportar uma língua local específica. O software interativo pode conter dezenas ou centenas de *prompts*, exibições de status, mensagens de ajuda, mensagens de erro e assim por diante. Os processos de projeto e construção devem acomodar problemas de sequência de caracteres, incluindo qual conjunto de caracteres deve ser usado, que tipos de sequências de caracteres são usados, como manter as sequências de caracteres sem alterar o código e traduzir as sequências em diferentes idiomas com impacto mínimo sobre o código de processamento e a interface do usuário.

9. **Processamento de entrada baseado em gramática:** o processamento de entrada baseado em gramática envolve a análise de sintaxe ou análise do fluxo de *token* de entrada. Inclui a criação de uma estrutura de dados (chamada árvore de análise ou árvore de sintaxe), que representa os dados de entrada. O analisador verifica a tabela de símbolos para a presença de variáveis definidas pelo programador que populam a árvore. Depois de construir a árvore de análise, o programa a usa como entrada para os processos computacionais.
10. **Concorrência simultânea:** incluem semáforos e monitores. Um semáforo é uma variável protegida ou um tipo de dados abstratos que fornece uma abstração simples, mas útil, para controlar o acesso a um recurso comum por vários processos ou segmentos em um ambiente de programação concorrente. Um monitor é um tipo de dados abstratos que apresentam um conjunto de operações definidas pelo programador executadas com exclusão mútua. Um monitor contém a declaração de variáveis compartilhadas e procedimentos ou funções que operam sobre essas variáveis. A construção do monitor garante que somente um processo por vez esteja ativo dentro do monitor.
11. **Middleware:** middleware é uma classificação ampla para software que fornece serviços acima da camada do sistema operacional ainda abaixo da camada do programa aplicativo. O middleware pode fornecer contêineres de tempo de execução para componentes de sof-

tware para fornecer passagem de mensagens, persistência e um local transparente em uma rede. Middleware pode ser visto como um conector entre os componentes que usam o middleware.

- 12. Métodos de construção para software distribuído:** um sistema distribuído é uma coleção de sistemas de computador fisicamente separados, possivelmente heterogêneos, que são conectados em rede para fornecer aos usuários o acesso aos vários recursos que o sistema mantém. A construção de software distribuído distingue-se da construção de software tradicional por questões como paralelismo, comunicação e tolerância a falhas.
- 13. Construindo sistemas heterogêneos:** os sistemas heterogêneos consistem em uma variedade de unidades computacionais especializadas de diferentes tipos, como microcontroladores e processadores periféricos. Essas unidades computacionais são controladas independentemente e se comunicam entre si. Sistemas embutidos são tipicamente sistemas heterogêneos. O sistema heterogêneo pode exigir a combinação de várias linguagens de especificação para projetar diferentes partes do sistema. Durante o código de hardware, o desenvolvimento de software e o desenvolvimento de hardware prosseguem concorrentemente por meio de decomposição por etapas. A parte de hardware é geralmente simulada em matrizes de portas programáveis em campo ou em circuitos integrados específicos de aplicações (ASICs). A parte do software é traduzida para uma linguagem de programação de baixo nível.
- 14. Análise de desempenho e ajuste:** a eficiência do código influencia a velocidade da execução. A análise de desempenho é a investigação do comportamento de um programa usando a informação recolhida à medida que o programa é executado, com o objetivo de identificar possíveis pontos quentes no programa a serem melhorados. O ajuste de código geralmente envolve apenas mudanças em pequena escala que afetam uma única classe, uma única rotina

ou, mais comumente, algumas linhas de código. Um conjunto de técnicas de ajuste de código está disponível, incluindo aqueles para sintonizar expressões lógicas, loops, transformações de dados, expressões e rotinas.

- 15. Padrões de plataforma:** os padrões de plataforma permitem que os programadores desenvolvam aplicações portáteis que podem ser executadas em ambientes compatíveis sem alterações. Padrões de plataforma normalmente envolvem um conjunto de serviços padrão e APIs que as implementações de plataforma compatíveis devem implementar.

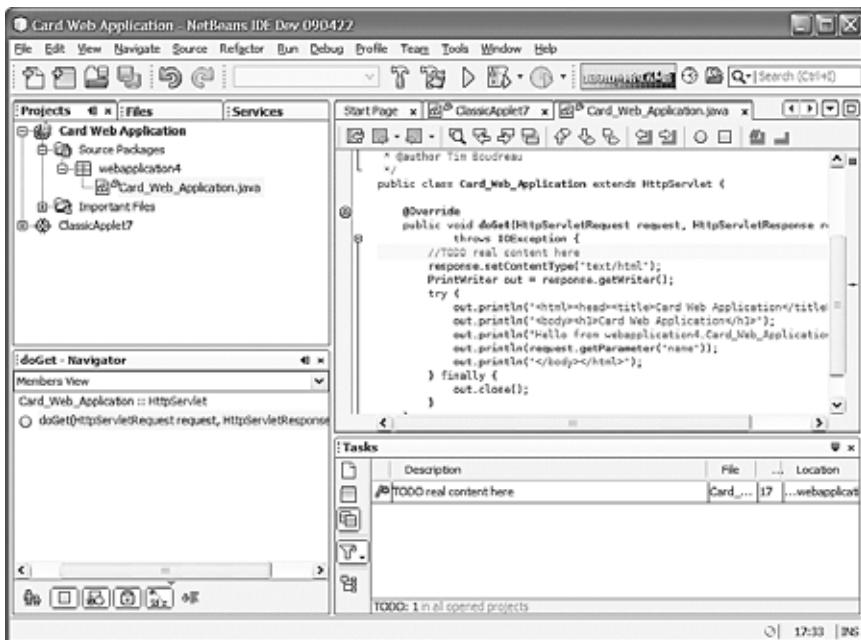
Exemplos típicos de padrões de plataforma são o Java 2 *Platform Enterprise Edition* (J2EE) e o padrão POSIX para sistemas operacionais (*Portable Operating System Interface*), que representa um conjunto de padrões implementados principalmente para sistemas operacionais baseados em UNIX.

4.5 Ferramentas de construção de software

- 1. Ambientes de desenvolvimento:** um ambiente de desenvolvimento integrado IDE (*Integrated Development Environment*) fornece instalações completas para desenvolvedores de software, integrando um conjunto de ferramentas de desenvolvimento. As escolhas dos ambientes de desenvolvimento podem afetar a eficiência e a qualidade da construção do software. Além das funções básicas de edição de código, os IDEs atuais geralmente oferecem outros recursos, como compilação e detecção de erros no editor, integração com controle de código-fonte, ferramentas de compilação, teste e depuração, visualizações compactadas de programas, transformações automatizadas de código e suporte para refatoração.

Alguns exemplos de IDEs: NetBeans, Visual Studio, Eclipse, JBuilder, Kdevelop, Brackets, Squad, Aptana, Dreamweaver.

Figura 4.2 – NetBeans: exemplo de IDE



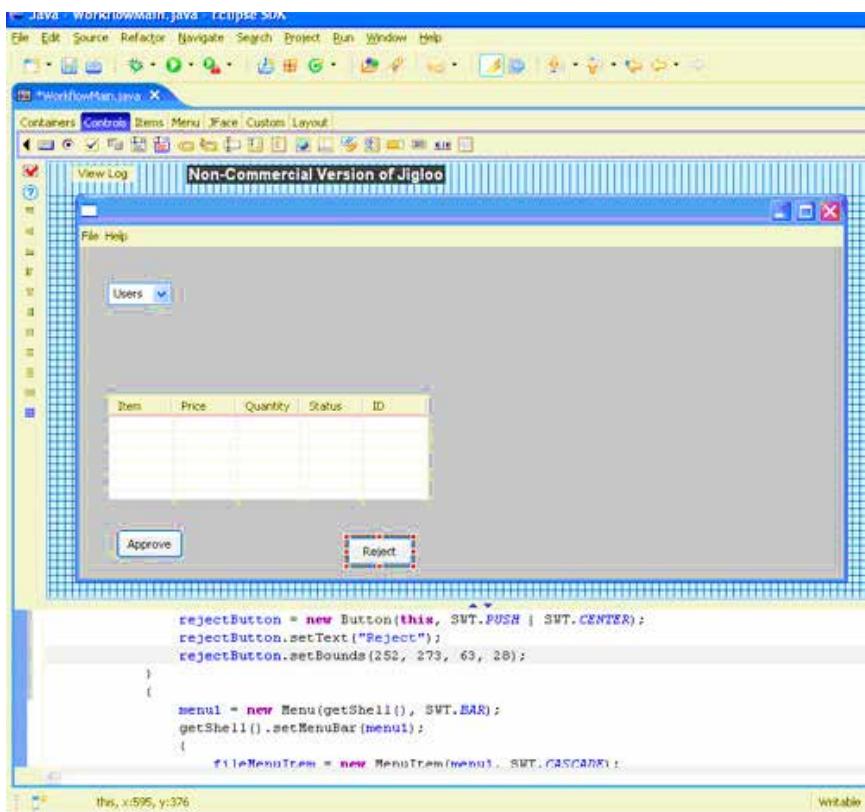
2. Construtores GUI: um construtor GUI (*Graphical User Interface*) é uma ferramenta de desenvolvimento de software que permite que o desenvolvedor crie e mantenha GUIs em um modo WYSIWYG (*What You See Is What You Get* ou O Que Você Vê é o Que Você Obtém). Um construtor de GUI geralmente inclui um editor visual para o desenvolvedor para criar formulários e janelas e gerenciar o layout dos *widgets* (componentes de interface) arrastar, soltar e configuração de parâmetro. Alguns construtores GUI podem gerar automaticamente o código-fonte correspondente ao design gráfico visual.

Como os atuais aplicativos GUI geralmente seguem o estilo orientado a eventos (no qual o fluxo do programa é determinado por eventos e manipulação de eventos), as ferramentas do construtor de GUI geralmente fornecem assistentes de geração de código que automatizam as tarefas mais repetitivas necessárias para o trata-

mento de eventos. O código de suporte conecta *widgets* com os eventos de saída e de entrada que acionam as funções que fornecem a lógica do aplicativo. Alguns IDEs fornecem construtores de GUI integrados ou *plug-ins* de construtor de GUI, como NetBeans, Eclipse e no JBuilder.

Alguns exemplos de construtores de GUIs: Jigloo, Window Builder e VE.

Figura 4.3 – Jigloo: exemplo de Construtor de GUI

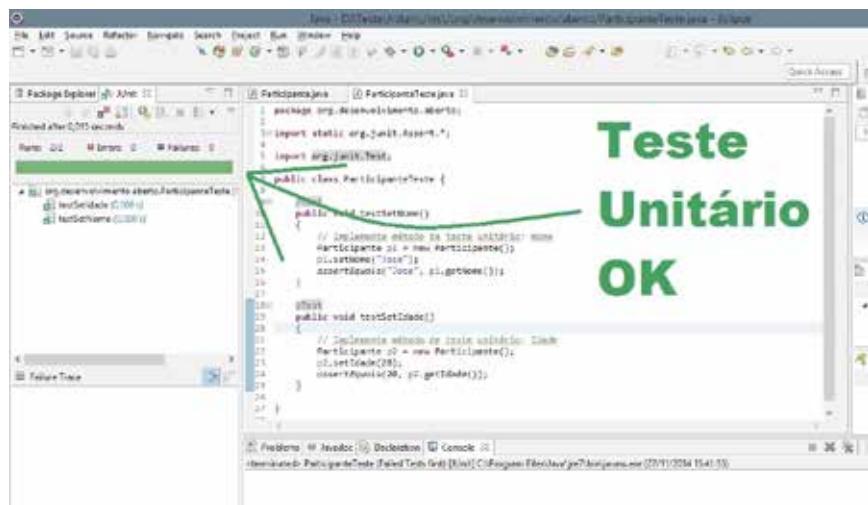


- 3. Ferramentas de teste unitário:** o teste unitário verifica o funcionamento de módulos do software isoladamente de outros elementos que são testáveis separadamente (por exemplo: classes, rotinas e

componentes). O teste unitário é, muitas vezes, automatizado. Os desenvolvedores podem usar ferramentas de teste unitário e estruturas para estender e criar ambiente de teste automatizado. Com ferramentas e estruturas de teste unitário, o desenvolvedor pode codificar critérios no teste para verificar a correção da unidade em vários conjuntos de dados. Cada teste individual é implementado como um objeto e durante a execução do teste, os casos de teste com falha serão automaticamente sinalizados e relatados.

Alguns exemplos de ferramentas de testes unitários: JUnit, JSUnit, CUnit, CPPUnit, PHPUnit e vPyUnit.

Figura 4.4 – JUnit framework: exemplo de ferramenta para automação de testes unitários

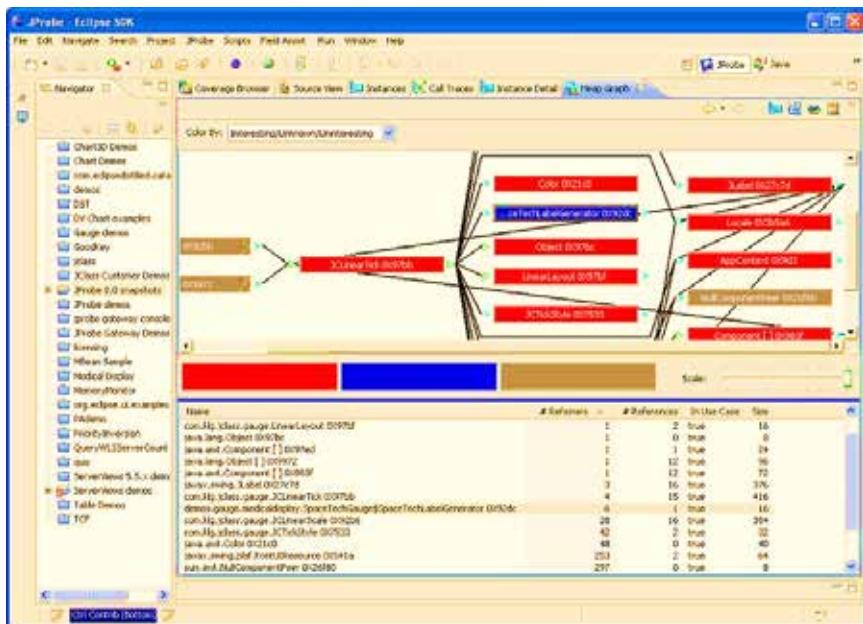


4. **Ferramentas de análise de desempenho:** são ferramentas normalmente usadas para suportar o ajuste de código. As ferramentas de análise de desempenho mais comuns são de criação de “*profilers*”. O *profiler* é usado para descrever o processo de medição do tempo de execução de métodos, para que assim possamos localizar e corrigir gargalos de desempenho. Uma ferramenta de *profiler* de execução monitora o código enquanto ele é executado e registra quantas vezes cada instrução é executada ou quanto tempo o programa

gasta em cada instrução ou caminho de execução. Isso dá uma visão de como o programa funciona e onde os desenvolvedores devem focar os esforços de ajuste de código.

Alguns exemplos de ferramentas de análise de desempenho: JProbe, JaViz, Hyperprof, ProfileViewer, Optimizelt, Visual Quantify e Hprof.

Figura 4.5 – JProbe Suite: exemplo de ferramentas de análise de desempenho



4.6 Introdução de boas práticas na codificação

A escolha da linguagem de programação apropriada que suporte os requisitos de projeto é um grande passo na direção da obtenção de um software de qualidade. No entanto, um fator de suma importância para a obtenção de um código claro e que ofereça facilidade de manutenção é a boa utilização dos mecanismos presentes na linguagem adotada. Na sequência, serão apresentados os principais fatores determinantes à obtenção de código-fonte bem escrito.

1. Documentação do código: o código-fonte não deve ser considerado como um resultado intermediário irrelevante no processo de desenvolvimento de um software. Dessa forma, a documentação é um elemento que deve ser seriamente considerado na etapa de codificação. Ela é um item essencial tanto para atividades de validação do software quanto para as tarefas de manutenção. Um primeiro passo na documentação do código-fonte é a escolha dos identificadores de variáveis e procedimentos. O uso de “siglas” ou caracteres para identificar os elementos de um programa é uma tendência herdada das linguagens de segunda geração que não encontra mais lugar nos dias atuais. Dessa maneira, é importante que os identificadores escolhidos tenham significado explícito para a aplicação considerada.

Outro elemento bastante considerado é a introdução de comentários ao longo do código-fonte, que pode ser bastante útil se utilizado eficientemente, podendo transformar-se no principal aliado às tarefas de manutenção. Uma sistemática interessante de uso dos comentários é a compatibilização destes com a documentação de projeto do software.

Um último aspecto importante na documentação é a formatação do código-fonte. Um mecanismo importante nesse contexto é o uso da indentação como meio de posicionar as instruções no contexto de trechos do código. A indentação permite explicitar melhor a combinação dos blocos básicos de uma linguagem de programação para a composição de componentes mais complexos.

- 2. Declaração de dados:** essa nem sempre é objeto de preocupação por parte dos desenvolvedores, porém à medida que o desenvolvimento vai se tornando complexo no que diz respeito à definição de estruturas de dados, o estabelecimento de uma sistemática padronizada de declaração de tipos de dados e variáveis vai assumindo um nível cada vez maior de importância.
- 3. Construção de instruções:** o fluxo lógico de instruções é definido normalmente durante a etapa de projeto. Entretanto, o mapeamento desse fluxo lógico em instruções individuais faz parte do

trabalho de codificação. Como discutimos anteriormente, um aspecto que deve ser explorado durante a construção é minimização da complexidade. Escrever mais de uma instrução por linha, por exemplo, é uma decisão que vai contra a clareza do código, independentemente da economia de espaço que isso pode representar. A seguir, são apresentadas algumas regras importantes no que diz respeito a esse aspecto.

- ✗ Evite o uso de testes condicionais complicados ou que verifiquem condições negativas.
- ✗ Evite o intenso aninhamento de laços ou condições.
- ✗ Utilize parênteses para evitar a ambiguidade na representação de expressões lógicas ou aritméticas.
- ✗ Utilize símbolos de espaçamento para esclarecer o conteúdo de uma instrução.

4. Entradas/Saídas: as entradas de dados em um software podem ser realizadas de maneira interativa ou em *batch*. Independentemente da forma como as entradas serão efetuadas, é importante considerar algumas regras:

- ✗ validar todas as entradas de dados;
- ✗ simplificar e padronizar o formato da entrada;
- ✗ estabelecer um indicador de final de entrada (ENTER, ponto final, etc.);
- ✗ rotular as entradas de dados interativas, indicando eventuais parâmetros, opções e valores *default*;
- ✗ rotular relatório de saída e projeto.

Síntese

Neste capítulo, foi visto que a construção de software está relacionado a implementação do software, verificação, testes unitários, teste de integração e testes de sistema. Nessa fase, o engenheiro de software deve minimizar a

complexidade, antecipar a mudança, pensar em construir para a verificação, utilizar o reuso e aplicar padrões em construção.

Vimos como a escolha dos modelos de ciclo de vida do software pode afetar a fase da construção do software e também verificamos que a construção é uma atividade em que o engenheiro de software tem de lidar com restrições do mundo real, algumas das quais discutimos, considerando a prática da construção.

Por fim, foram apresentadas algumas ferramentas de construção de software utilizadas pelo mercado, bem como alguns pontos-chave para aplicação de boas práticas na codificação, como o uso de documentação do código, declaração de dados, construção de instruções e entradas/saídas.

Atividades

1. O que significa o termo “reuso” na construção de software? Explique a diferença entre “construção para reutilização” e “construção com reutilização”.
2. Por que o uso de padrões em construção é importante? Justifique sua resposta.
3. Explique o objetivo do teste de construção e quais os tipos mais comuns.
4. A documentação do código-fonte é necessária? Explique por quê.

5

Testes de Software

NÃO HÁ COMO falar sobre Engenharia de Software sem vincular à qualidade de software e não há como falar em qualidade de software sem mencionar testes de software. O teste de software tornou-se parte importante no desenvolvimento de software e cada vez mais sua “fazia” de representatividade e importância nos projetos de desenvolvimento de software vem aumentando.

NESTE CAPÍTULO, ABORDAREMOS os testes de software, apresentando a importância que ganhou com o passar dos anos, transformando-se de atividade subjugada para atividade primordial nas organizações que constroem softwares para uso próprio ou para uso comercial.

5.1 Fundamentação de teste de software

Ao longo dos anos, os testes ganharam importância dentro dos processos de desenvolvimento de software. Na década de 1950, entendia-se como teste uma atividade para indicar que o software funcionava; já em 1979, após a publicação do livro *The art of software testing*, de Glenford Myers, o conceito foi ampliado. O autor foi um dos pioneiros na área, apresentando técnicas e estudos sobre a economia gerada pelos testes de software e trazendo a visão de que testes são atividades realizadas com a intenção de encontrar erros, e não apenas validar o funcionamento correto dos sistemas. Por volta da década de 1990, o conceito evoluiu e entendeu-se que testes são importantes para gerenciar a qualidade.

Em face da evolução tecnológica que permitiu a redução de custos e o aumento da produção de softwares como um nicho muito promissor financeiramente, o mercado de software estava em constante busca de melhoria no processo. Fatores como a globalização da economia e o aumento da competitividade do mercado fizeram crescer a demanda por aplicações que facilitassem os processos operacionais das empresas tornando-os mais ágeis. Isso provocou o desenvolvimento de softwares mais complexos, com integração em mais de um ambiente e inúmeros componentes que se comunicam entre si. Contudo, muitos softwares comercializados têm apresentado vários defeitos, gerando alto grau de insatisfação dos usuários, prejuízos financeiros, perda de participação no mercado ou danos para a imagem do produto.

Informações de mercado dizem que mais de 90% dos sistemas são liberados com graves defeitos. Softwares com problemas de performance e com defeitos na execução são custosos. Informações atualizadas deste estudo apontam para o mesmo resultado, mostrando que muito pouco foi alterado na situação vigente da época. (RIOS, 2013, p. 1)

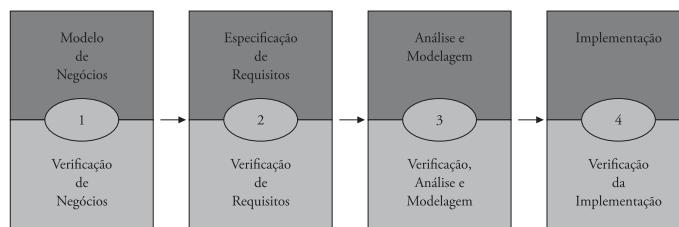
Dessa forma, as empresas de desenvolvimento de software têm o desafio contínuo de produzir sistemas cada vez mais de acordo com as tendências tecnológicas e as necessidades do mercado, especialmente com prazos curtos, custos razoáveis e qualidade. Em razão disso, as atividades de teste de software estão sendo utilizadas cada vez mais para evitar que os problemas e as deficiências contidas no software desenvolvido sejam sanados antes que o produto seja entregue para o cliente.

Podemos entender que a atividade de teste de software apresenta os seguintes focos: encontrar falhas, prevenir falhas e evitar falhas. Sendo assim,

o teste abrange muito mais que uma simples execução ou utilização de produto. No caso do software, engloba um conjunto de atividades planejadas e controladas, que requerem conhecimentos de técnicas e métodos de teste específicos. Ele pode ser trabalhado com dois enfoques: verificação e validação do software. Além disso, uma análise dinâmica do software faz parte da atividade, ou seja, é preciso utilizar o software com o objetivo de verificar a presença de falhas a fim de diminuir os erros e aumentar a confiança da conformidade com os requisitos do sistema.

- ✖ Verificação se refere ao conjunto de atividades que fazem a checagem da consistência do software com a especificação ou a documentação de requisitos. É um monitoramento realizado em todas as fases de desenvolvimento e uma garantia de que o software está sendo construído corretamente, conforme mostra a figura 5.1.

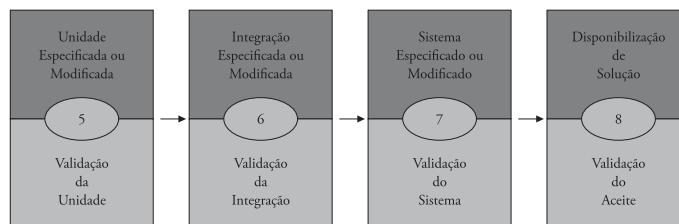
Figura 5.1 - Etapas dos testes de verificação



Fonte: Bartié (2002).

- ✖ Validação é o conjunto de atividades que avaliam em que grau o software construído satisfaz às necessidades reais do usuário, ou seja, visa garantir se o produto certo foi construído, como mostra a figura 5.2.

Figura 5.2 - Etapas dos testes de validação



Fonte: Bartié (2002).

Independentemente do enfoque dado ao teste, é necessário estabelecer os atributos de qualidade do software que devem ser considerados no planejamento de testes. Com base na norma ISSO/IEC 9126, é possível trabalhar com seis atributos-chave de qualidade requerida para um software.

1. **Funcionalidade:** diz respeitos às funções definidas para o software. Nesse item, analisa-se se o software funciona adequadamente e atende aos requisitos especificados;
2. **Confiabilidade:** indica que software deve funcionar por determinado período de tempo livre de erros e falhas;
3. **Usabilidade:** mostra se o software é compreendido e utilizado facilmente pelo usuário;
4. **Eficiência:** identifica se o software atinge um excelente desempenho utilizando seus recursos da melhor forma;
5. **Manutenibilidade:** indica se é possível realizar alterações e correções no software sem grandes dificuldades;
6. **Portabilidade:** mostra a capacidade de o software ser transferido facilmente de um ambiente (plataforma) para outro.

Existem outros atributos que podem ser utilizados como base para se ter um software de qualidade e dentre eles pode-se destacar a importância de ter profissionais preparados e experientes, metodologias e ferramentas refinadas, participação constante dos clientes e compreensão correta do problema com modelagem flexível de longo prazo (BARTIÉ, 2002).

Para melhor entendimento da função e da importância dos testes dentro da Engenharia de Software, convém inicialmente definir o conceito de defeito, erro e falha. Segundo Pressman (2011), defeito e falha são vistos como sinônimos, pois indicam um problema de qualidade descoberto depois que o software foi entregue ao cliente; já o termo erro trata de um defeito detectado e solucionado antes de o software ser liberado para o cliente.

No entanto, nem todos concordam com essa definição, que toma como base o momento em que os problemas são conhecidos. De acordo com o IEEE (Institute of Electrical and Electronics Engineers – Instituto de Engenheiros Eletricistas e Eletrônicos, em português), há uma clara distinção entre os termos:

- ✖ defeito está relacionado a documentação de requisitos, código fonte, modelos de dados. Ocorre na tentativa de o indivíduo entender determinada informação, resolver um problema, utilizar um método ou uma ferramenta. Isso significa que o defeito não depende de um programa para existir, podendo ser identificado antes mesmos dos testes e da codificação. Por exemplo: requisitos ambíguos ou incorretos e comando incorreto;
- ✖ erro (*bug*) está relacionado a resultados da execução de um programa e diferença entre o valor retornado e o valor esperado, ou seja, quando o software apresenta um resultado inesperado;
- ✖ falha se refere a um resultado operacional do software diferente do que foi especificado ou esperado pelo usuário. Uma falha pode ser causada por diversos erros e alguns erros podem nunca causar uma falha.

De modo geral, defeito e erro são referenciados como a causa; e a falha, como a consequência a um comportamento inadequado do programa. Vale a pena esclarecer que *bug* é uma palavra de origem inglesa que significa inseto e se popularizou na informática com a divulgação da história do Eniac (Electronic Numeral Integrator Analyzer and Computer – Computador Integrador Numérico Eletrônico, em português), um dos primeiros computadores eletrônicos, uma máquina gigantesca que ocupava uma sala inteira e que foi desenvolvido na época da Segunda Guerra Mundial. Era composto por milhares de resistores e válvulas e a luz desses componentes atraía muitos insetos. Toda vez que algum resistor ou válvula queimava, provocando o mau funcionamento do computador, vários insetos estavam acumulados no local, vindo daí o vínculo da palavra *bug* com erro do software.

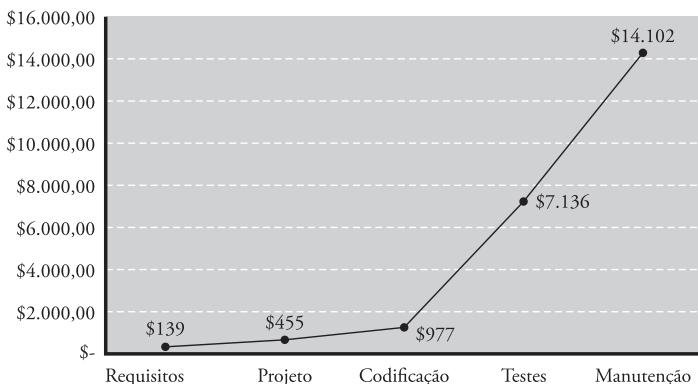
Com base nessas definições, é fácil compreender que muitos dos defeitos encontrados no código da aplicação são originados na fase de levantamento de requisitos, e outros são introduzidos durante a codificação do software. Por meio da atividade de testes, é possível avaliar a qualidade do software em todo o processo de desenvolvimento. Porém, mesmo com esses testes, muitos dos defeitos já convertidos em falhas são percebidos apenas pelos usuários finais.

Uma correção de defeito surgido no ambiente de produção pode chegar a ser cem vezes mais cara do que se essa correção fosse realizada nas fases

iniciais do projeto. Dessa forma, quanto antes for identificado um defeito ou uma falha, mais barata será sua correção.

Segundo Pressman (2011), “o custo da qualidade inclui todos os custos necessários para a busca de qualidade ou para a execução de atividades relacionadas à qualidade, assim como os custos causados pela falta de qualidade”.

Figura 5.3 – Custo relativo para correção de erros e de defeitos



Fonte: Presman (2011).

De acordo com os estudos publicados pela sexta edição do *World Quality Report* (Relatório Mundial sobre Qualidade), haverá um crescimento nos investimentos em testes de aplicativos que devem chegar a 29% em 2017 (WORLD..., 2014). Testes podem apresentar custos elevados, mas devem ser encarados como investimento em qualidade, uma vez que contribuem bastante para melhorar a qualidade do software, evitar o retrabalho, aumentar a segurança e a confiabilidade, reduzir custos e revelar falhas e incompatibilidades e com isso evitar impactos negativos para os clientes.

5.1.1 Atividade de testes

Segundo Pressman (2011), testes não são capazes de mostrar a ausência de erros e defeitos, apenas mostrar que erros e defeitos de software estão presentes. É importante destacar essa percepção, pois, quando os testes não identificam defeitos, não significa necessariamente que o software é um produto de boa qualidade. Ocorre que essa situação pode ter acontecido porque o

teste foi realizado sem planejamento ou sem critérios e não foi suficiente para revelar os erros existentes, sendo, portanto, um teste de baixa qualidade.

Essa é uma atividade que deve ser aplicada logo no início do processo de desenvolvimento do software e realizada simultaneamente durante o processo até a entrega ao usuário, para que os resultados de qualidade sejam atingidos (RIOS, 2008). A partir dessa afirmação, entende-se que o teste não é uma tarefa simples e exige planejamento e conhecimento, por isso sua aplicação se depara com dificuldades, tais como Crespo e Jino (2005) apontam:

- ✗ **custos** – o teste de software envolve de fato certo custo, há demonstrações em diversas pesquisas de que o teste de software responde de 30% a 50% dos custos totais do desenvolvimento do software. Contudo, tirando o fator monetário, existe o fator psicológico, pois muitos o veem como a causa do aumento dos custos e dos prazos do projeto;
- ✗ **atividade pouco valorizada** – falta de conhecimento sobre a relação custo/benefício do teste;
- ✗ **profissionais** – poucos profissionais especializados na área de teste, a maioria desenvolvedores, e o perfil para a área requer conhecimentos e habilidades diferenciadas;
- ✗ **falta de conhecimento e preparo** – sem o conhecimento adequado sobre o processo de teste de software, das técnicas e de seus critérios, das fases e dos tipos de testes, fica difícil criar um ambiente adequado, um planejamento e uma preparação para a execução dos testes, o que propicia que muitos erros não sejam encontrados, resultando em um software com baixa qualidade.

A essência do teste é, portanto, identificar e revelar as fraquezas do software, fornecendo informações a respeito da qualidade dele. É uma atividade de investigação e de análise com uma abordagem contínua da qualidade desde o início do processo de desenvolvimento do software.

Vale ressaltar que a atividade de teste é pautada em alguns princípios definidos por Glen Myers e Davis (PRESSMAN, 2011):

- ✗ o objetivo dos testes é executar o programa visando encontrar erros;

- × um bom pacote de testes é aquele que tem alta probabilidade de encontrar um erro ainda não conhecido;
- × um teste de sucesso é aquele que revela um novo erro;
- × o teste não é capaz de provar a ausência de erros;
- × os testes devem estar alinhados com os requisitos do cliente;
- × os testes devem ser planejados antes de serem iniciados;
- × os testes devem iniciar nos componentes individuais e, à medida que progredem, devem se concentrar nos componentes integrados e em seguida no sistema inteiro;
- × é impossível realizar testes completos em sistema, seja de qual porte for; é exaustivo e impossível executar todas as combinações de rotas durante o teste;
- × é importante realizar testes estáticos; mais de 85% de defeitos do software são originados na documentação;
- × o total de defeitos descobertos é um bom indicador da qualidade do software e os tipos de defeitos encontrados podem ser uma medida da estabilidade do software;
- × testes devem ser realizados por terceiros, e não por aqueles que produziram o código; h uma chance muito maior de erros serem identificados quando os testes são realizados por outras pessoas.

5.2 Processo de testes de software

Realizar testes da maneira adequada requer controle e organização, e por isso precisa ser estruturado em etapas, atividades, artefatos, papéis e responsabilidades, visando a padronização dos trabalhos. O processo de teste, de maneira geral, adapta-se ao processo e ao modelo de desenvolvimento de software utilizado pela organização. No entanto, o processo de teste de software é composto basicamente das seguintes etapas:

- × **planejamento** – proposta de testes que procurando atender aos quesitos de qualidade do cliente, dimensionando os esforços necessários;

sários para os testes, como equipe, prazos, custos, ferramentas de testes, hardware, tipos, níveis e técnicas de testes que serão utilizados, além dos impactos, dos riscos e das contingências para atividades de testes; nessa etapa são definidos os critérios para determinar quando a atividade estará completa;

- ✖ **configuração** – preparação do ambiente que será utilizado para os testes, tais como hardwares, softwares e massa de dados;
- ✖ **projeto de casos de testes** – principal atividade dentro do processo de testes, indicará se o teste será bem-sucedido no quesito de encontrar defeitos com o menor esforço possível. Para construir bons casos de testes, existe uma série de técnicas para ajudar nessa etapa, já que é nesse momento que os casos de testes serão projetados e devem indicar as funcionalidades a serem testadas e devem ser identificados e definidos os critérios de passagem e falha por funcionalidade (RIOS, 2008);
- ✖ **execução dos testes** – transformação dos casos de testes em forma executável, simulando a ação dos usuários no software. Essa execução pode ser realizada de forma manual ou automatizada (nesta, é necessária uma ferramenta para execução); além disso, é nessa etapa que os resultados encontrados devem ser registrados e comparados com os resultados esperados;
- ✖ **avaliação dos resultados** – análise das falhas identificadas e de seus impactos com geração de indicadores e relatórios; é interessante que, caso não tenham sido identificadas falhas, os casos de testes sejam revisados e se defina por encerrar ou não a atividade de testes.

É importante que cada uma das etapas citadas seja documentada. Segundo Rios (2008), não é possível que o processo de teste funcione adequadamente se não tiver documentos padronizados.

5.3 Níveis de teste de software

De acordo com Rocha (2001), os principais níveis de teste de software são:

- × **teste de unidade ou teste unitário** – tem como função explorar os menores elementos testáveis do software com o objetivo de encontrar falhas causadas por defeitos de lógica e de implementação em cada unidade separadamente. O conteúdo de análise são métodos de objetos, módulos, funções ou mesmo pequenos trechos de código. Esses testes são realizados pela pessoa que construiu o código (o desenvolvedor) à medida que a unidade é desenvolvida;
- × **teste de integração** – tem como objetivo provocar falhas decorrentes da integração entre uma ou mais unidades de código ou componentes do software associados com as interfaces que construirão a estrutura do software. Busca validar se as unidades que foram testadas individualmente executam corretamente quando integradas. Recomenda-se que seja a equipe de teste a responsável por executar testes nesse nível, no qual é possível adotar as seguintes estratégias:
 - × **teste não incremental** – os testes são realizados com os elementos de software combinados todos de uma vez, ao invés de estágios, ou seja, o programa inteiro é testado de uma só vez. Essa estratégia também é conhecida como big-bang. Não é a mais recomendada pelo fato de ser difícil isolar as causas das falhas encontradas;
 - × **teste incremental** – dividido em dois tipos:
- × **integração descendente ou top-down** – os testes são realizados de cima para baixo, iniciando pelo nível hierárquico superior do software (módulo principal), que geralmente é o módulo que controla todos os demais. A princípio é testado sozinho e, em seguida, em combinação com os módulos que são chamados por ele, e isso se repete até que seja realizado teste com o sistema inteiro;
- × **integração ascendente ou bottom-up** – é exatamente o contrário da descendente, e os testes são realizados de baixo para cima, ou seja, testa-se individualmente o componente mais inferior da hierarquia do software até que todos os módulos tenham sido testados. Ao final, todos os módulos são testados juntos.
- × **teste de sistema** – busca falhas por meio da execução do software, utilizando-o como se fosse um usuário final, passando por todas as

funcionalidades. Deve simular o ambiente de execução do usuário e, para isso, os testes devem ser executados em ambiente similar ao dos futuros usuários e com os mesmos dados que serão manipulados no dia a dia. O objetivo é descobrir erros de função e características de desempenho que não atendam aos requisitos ou à especificação e deve ser conduzido pela equipe de testes;

- × **teste de aceitação** – é o teste final, realizado antes da liberação do software para os clientes. Geralmente é conduzido por um pequeno grupo de usuários finais do sistema que o executam simulando operações de rotina para verificar se está pronto para ser utilizado e atende às necessidades para o qual foi solicitado. O teste de aceitação pode ser:
 - × formal – gerenciado, planejado e com critérios de aceitabilidade, com uma seleção de casos de testes rigorosos. Nessa forma, as funções que serão testadas, os detalhes dos testes e os resultados são conhecidos e controlados;
 - × informal – não é gerenciado nem possui um controle, sendo mais subjetivo. Existe um planejamento, mas de forma simplificada. Nesse caso, não há uma seleção de casos de testes específicos para serem executados, sendo a pessoa que executa o teste quem determina o que será realizado.

O planejamento pode ser estruturado conforme cada fase do processo de desenvolvimento do software. Assim como existem níveis de teste para ajudar no planejamento, existem técnicas de teste de software que podem ser aplicadas nos diferentes níveis de testes.

5.4 Técnicas de teste de software

O teste de software deve buscar avaliar todos os aspectos do desenvolvimento de um software. Além de avaliar questões comportamentais, de design, desempenho e de funções, deve se preocupar também com questões como integridade dos dados, capacidade de ser executado e instalado em diferentes configurações, capacidade de lidar com grande quantidade de solicitações ao mesmo tempo (teste de carga) e confiabilidade em casos de baixo processa-

mento, como baixa memória, serviços e hardwares indisponíveis ou recursos limitados (teste de estresse). Para isso, existem técnicas diferentes de testes que devem ser executadas e planejadas na avaliação do software. Cada técnica possui um foco e objetivo específico.

Atualmente, há uma diversidade de técnicas que podem ser aplicadas para o teste de software e que cobrem diferentes perspectivas do software, classificadas segundo os objetivos de teste. As mais conhecidas são teste funcional e teste estrutural.

Cada uma delas apresenta diversos critérios de teste. Embora tenham objetivos diferentes, uma não descarta a outra, pois há situações em que o teste funcional não é capaz de identificar alguns defeitos, já que essa técnica se preocupa em comparar se as saídas produzidas pelo software estão conforme o resultado esperado e não considera a funcionalidade interna do software. Em razão disso, dentro do código do software podem existir defeitos que não afetam as respectivas saídas e que não serão descobertos se apenas o teste funcional for utilizado (FURLAN, 2009).

5.4.1 Teste funcional

Também conhecido como teste de caixa preta ou teste de comportamento, possui esse nome por considerar o software como uma caixa preta da qual só se conhece a parte externa, mas não a parte interna (MEIRELLES, 2008). O teste funcional concentra-se nos requisitos funcionais documentados pela especificação do programa, e não em detalhes internos do código. Essa técnica pode ser aplicada a qualquer descrição comportamental do programa, em qualquer nível, desde o teste de unidade até o de sistema e é mais barato de projetar e executar do que o teste estrutural (PEZZÈ; YOUNG, 2008).

As técnicas de teste de caixa-preta permitem verificar séries de entradas que utilizarão completamente todos os requisitos funcionais. Com elas é possível detectar erros de funções incorretas ou ausentes, erros de interface, erros nas estruturas de dados ou no acesso à base de dados externas, erros de comportamento ou desempenho e erros de iniciação e término.

Segundo Pressman (2011), esse teste procurar responder às seguintes perguntas.

- ✗ Como a validade funcional é testada?
- ✗ Como o comportamento e o desempenho do sistema são testados?
- ✗ Quais classes de entrada farão bons casos de teste?
- ✗ O sistema é particularmente sensível a certos valores de entrada?
- ✗ Como as fronteiras de uma classe de dados são isoladas?
- ✗ Quais taxas e quais volumes de dados o sistema pode tolerar?
- ✗ Que efeito combinações específicas de dados terão sobre a operação do sistema?

Para Sommerville (2011), o teste funcional possui dois passos principais: identificar todas as funções definidas para o software realizar e projetar casos de teste que sejam capazes de validar se todas as funções estão sendo realizadas pelo software. Por depender totalmente da especificação do programa, esse tipo de teste é mais completo e eficiente se a especificação for bem elaborada e estiver de acordo com os requisitos do cliente. Por isso, quanto mais conhecimento o analista responsável por realizar esse tipo de teste tiver, melhores serão os casos de teste. Segundo Bartié (2002), esse profissional deve ter “conhecimento dos requisitos, suas características e comportamentos esperados, para que seja possível avaliar o software através dos resultados gerados pela aplicação”.

De forma objetiva, o teste funcional realiza a verificação da conformidade entre o produto implementado e os requisitos funcionais. Basicamente, o testador apresenta as entradas ao sistema e valida as saídas. Se as saídas forem diferentes das previstas na especificação do programa, o teste é considerado um sucesso, pois identificou uma falha no software.

Essa técnica apresenta algumas vantagens: é extremamente útil quando o objetivo é encontrar defeitos de requisitos e também de implementação; facilita a avaliação dos resultados, já mesmos são visuais pela aplicação; possibilita que testes mais simples sejam projetados; e não exige um conhecimento da tecnologia empregada para que os testes sejam realizados. Mas também apresenta algumas desvantagens: convenciona-se a seguir um padrão de procedimentos de testes e demanda mais manutenção dos artefatos de testes gerados.

Complementando as vantagens e as desvantagens dessa técnica, vale ressaltar dois pontos já mencionados:

- ✗ é uma técnica que depende totalmente da documentação de desenvolvimento do software, portanto, é importante que a documentação tenha alta qualidade, o que torna isso uma desvantagem;
- ✗ pode ser aplicada em todos os níveis de testes, o que se considera uma vantagem.

É importante destacar também que a maioria dos testes funcionais pode ser automatizada. A seguir, uma lista de possíveis falhas que podem detectadas por essa técnica:

- ✗ funções incorretas ou que foram omitidas;
- ✗ falhas em interface, comportamento ou desempenho;
- ✗ falhas ao abrir ou fechar o programa;
- ✗ campos que não aceitam a quantidade de caracteres necessários;
- ✗ falta de campos importantes;
- ✗ campos que aceitam letras e símbolos quando deveriam aceitar somente números;
- ✗ dados duplicados;
- ✗ campos sem a devida consistência, como o CPF.

Devido ao teste funcional ter por objetivo validar as entradas e as saídas, é quase impossível validar todas as possibilidades de combinações de entradas válidas e inválidas, mesmo que para funções simples. Para ajudar nisso, essa técnica possui alguns critérios, como:

1. **particionamento de equivalência** – divide o domínio de entrada de dados de um módulo em uma série de diferentes classes ou partições de equivalência, cada uma delas representando um possível erro a ser descoberto. Dessa forma, reduz-se um grande ou infinito conjunto de entradas para um pequeno, mas eficiente conjunto de entradas.

Essas classes podem ser divididas em entradas válidas e não válidas, obtendo-se os casos de testes. Com isso, se um caso de teste bem

elaborado de cada classe detectar falha, todos os demais componentes dessa classe também apresentarão o mesmo problema. Para auxiliar na construção dos casos de testes com esse critério, existem algumas diretrizes:

- ✗ se uma condição de entrada especificar um intervalo de valores, deve ser definida uma classe de equivalência válida e duas inválidas. Por exemplo, o campo tem como requisito aceitar somente valores entre 1 e 99. O teste de equivalência válida seria entrar com valores 1 e 99 e o teste de equivalência inválida seria entrar com valores menores que 1 e maiores que 99;
- ✗ se uma condição de entrada exige um valor específico, deve ser definida uma classe de equivalência válida e duas inválidas. Por exemplo, se um campo tem como requisito receber valores numéricos, o teste de equivalência válida são números e o de equivalência inválida seria outros valores que não fossem números, como letras e caracteres especiais;
- ✗ se uma condição de entrada especificar um conjunto de valores, deve ser definida uma classe de equivalência válida e uma inválida. Por exemplo, o campo situação só pode receber os valores ativo, desativado e cancelado. O teste de equivalência válida são os valores especificados e o de equivalência inválida seriam todos os outros valores diferentes dos especificados. Cada valor especificado deve ser testado distintamente.
- ✗ A partição do evento de saída fica nas entradas do programa. Mesmo que classes diferentes de equivalência de entrada pudessem ter o mesmo tipo de evento de saída, ainda assim as classes de entrada seriam tratadas distintamente.

Esse critério apresenta como vantagem a redução e a diversificação do conjunto de casos de testes, porém a desvantagem é que não é fácil identificar todas as classes de equivalência.

2. análise do valor limite –

esse critério complementa o de particionamento de equivalência, que utiliza valores limites de entrada ou de saída, porém, nem sempre testes que exploram valores limite

dão mais retorno do que os testes que não o fazem. Segundo Bartié (2002), a análise do valor limite se diferencia do particionamento de equivalência em dois pontos:

- ✗ esse critério trabalha com entradas próximas aos limites ao invés de qualquer valor dentro dos limites de uma classe de equivalência, pois o software está mais suscetível ao erro nas condições limites dos domínios de dados do que propriamente nas regiões centrais;
- ✗ há uma preocupação não somente com as condições de entrada, mas também com as saídas.

Em cada classe de equivalência, a análise se concentra nas condições de limite nas quais se considera como tendo uma taxa maior de sucesso para encontrar defeitos do que nas condições sem limite. O limite ou fronteira de cada classe de equivalência são os valores em seu máximo e em seu mínimo, ou seja, os valores imediatamente “acima de” ou “abaixo de”. Faz uso dos valores: mínimo inválido, mínimo válido, máximo válido e máximo inválido. Portanto, é necessário ter mais testes do que no critério de particionamento.

3. **grafo de causa-efeito** – o primeiro passo é entender a ligação entre os objetos ou módulos do software e as relações que os conectam e com base nisso construir casos de testes que verificam se todos têm a relação esperada uns com os outros. Os critérios anteriores não exploram combinações dos dados de entrada, já o grafo de causa-efeito procura suprir essa deficiência. Nesse critério, utilizam-se tabelas de decisão e árvores de decisão.

Para utilizá-lo, é necessário primeiro identificar as causas (condições de entradas ou estímulos que provoquem uma resposta do software) e efeitos (saídas ou ações realizadas em resposta às diferentes condições de entrada) com base na especificação do software. Em seguida, é construído um grafo relacionando as causas e os efeitos e depois o grafo é convertido em uma tabela de decisão a partir de qual se constroem os casos de teste (PRESSMAN, 2011).

5.4.2 Teste estrutural

Também conhecido como teste de caixa branca ou de vidro em oposição à caixa branca, justamente porque considera a parte interna do software. Por razões óbvias, também é conhecido como teste baseado no código, pois baseia-se em um minucioso exame da estrutura de código e de dados. São testados os caminhos lógicos do software gerando casos de teste que põem à prova conjuntos específicos de condições e laços (SILVA, 2005).

Essa técnica de teste tem o objetivo de encontrar comandos incorretos, estruturas de dados e de programação incorretas, variáveis não definidas e erros de inicialização e finalização de *loops* (FURLAN, 2009).

Essa técnica avaliará teste de condição, teste de fluxo de dados, teste de ciclos, teste de caminhos lógicos e códigos nunca executados. Ela complementa a técnica de teste funcional, uma vez que permite identificar falhas que ainda não são possíveis de ser verificadas pela técnica de teste funcional.

Para realizar testes nessa técnica, o profissional deve ter conhecimento da tecnologia utilizada no software, além da arquitetura interna da solução, tendo de acessar as fontes e a estrutura de banco de dados (BARTIÉ, 2002).

Dessa maneira, tem-se a impressão de que, para ter um software livre de falhas, basta realizar testes com base nessa técnica. Mas não é bem assim, pois seriam necessários, mesmo nessa técnica, testes exaustivos para validar todas as combinações possíveis de caminhos e de valores de variáveis, o que seria impraticável. Essa técnica é útil quando se deseja garantir que todos os fluxos independentes tenham sido executados pelo menos uma vez.

Segundo Wazlawick (2013), essa técnica apresenta uma série de limitações e de desvantagens em relação à funcional:

- ✗ não permite identificar defeitos nas especificações;
- ✗ não verifica se o software se comporta de acordo com a especificação esperada;
- ✗ não necessariamente cobre problemas com estrutura de dados, como *arrays* e lista;

- ✗ não trata de situações típicas de programas orientados a objetos, tais como herança e polimorfismo;
- ✗ não consegue testar funcionalidades ausentes, já que preconiza o teste daquilo que existe no programa; por exemplo, se um comando deveria ter sido incluído no código, mas não foi;
- ✗ não pode ser utilizada com a técnica de desenvolvimento dirigido pelo teste (TDD).

Além dessas limitações, essa técnica é difícil de ser automatizada e não se aplica a todos os níveis de testes, ficando mais restrita ao nível de teste unitário. Da mesma forma que o teste funcional, para facilitar na projeção e nos casos de testes, essa técnica possui alguns critérios. Dentre os principais temos critérios baseados em fluxo de controle, fluxo de dados e de complexidade.

Os critérios baseados em fluxo de controle, como o próprio nome diz, focam as estruturas de controle de execução do programa, tais como comandos ou desvios, condições e laços, dos quais se derivam os requisitos de teste. Os critérios baseados em fluxo de dados, por sua vez, utilizam informações do fluxo de dados do programa, que se baseiam nas interações que envolvem definições de variáveis e referência a elas para derivar os requisitos de teste. Já os critérios baseados na complexidade derivam os requisitos de teste a partir da complexidade do programa (MALDONADO et al, 1998).

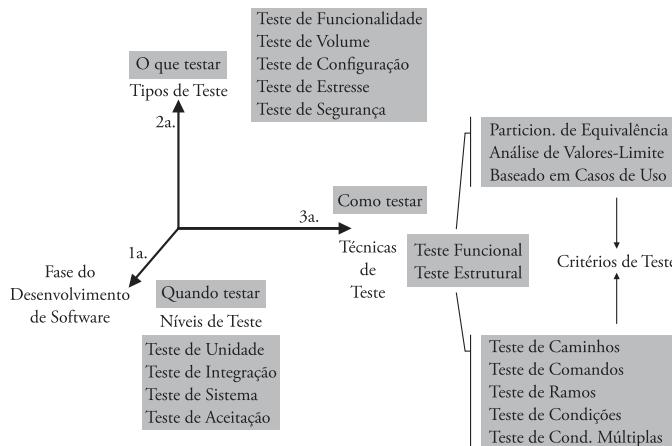
Além das técnicas de testes, existem também os tipos de testes. Há uma grande quantidade de tipos que podem ser utilizados. A seguir, temos um breve descritivo de alguns tipos mais conhecidos (PFLEEGER, 2004):

- ✗ **volume** – visa verificar a capacidade do software de lidar com grande quantidade de dados, seja de saída, seja de entrada;
- ✗ **configuração** – procura garantir que o software consiga funcionar adequadamente e de acordo com o esperado em diferentes configurações de software e/ou hardware;
- ✗ **estresse** – procura validar a confiabilidade do sistema, validando o comportamento do software quando submetido a condições anormais como: quando ocorrer uma quantidade máxima de usuários e

- dispositivos conectados a ele, memórias insuficientes, hardwares ou recursos indisponíveis ou limitados;
- ✖ **segurança** – valida se os dados do sistema são acessados apenas por aqueles que possuem permissão, procurando, dessa forma, validar o quanto o software mantém a integridade e a confidencialidade dos dados em casos de ataques para roubo de senhas, por exemplo.

A figura 5.4 permite uma compreensão melhor sobre a relação entre níveis, tipos, técnicas e critérios de teste.

Figura 5.4 - Relação entre níveis, tipos, técnicas e critérios de testes



Fonte: Crespo; Jino (2005).

5.5 Métricas de teste de software

Embora haja muitas maneiras de medir o sucesso de um sistema ou produto com base em computador, a satisfação do usuário está no topo da lista (PRESSMAN, 2011). Métricas de teste enquadram-se em duas categorias amplas: métricas que tentam prever o número provável de testes necessário nos vários níveis de teste e métricas que focalizam a cobertura de teste para dado componente (PRESSMAN, 2011).

As medidas básicas se obtêm diretamente dos dados reunidos pelo analista de teste e são utilizadas para acompanhamento do andamento do projeto, como a quantidade de casos de testes que foram criados, que foram executados, que foram bloqueados, que falharam, que passaram, que estão em andamento e que tiveram de ser reexecutados.

Já as métricas derivadas são obtidas pelo líder ou pelo gerente de testes, que converte as métricas básicas em dados combinados que geram informações a respeito do processo de teste, como percentual de testes concluídos, quantidade de falhas na primeira execução dos testes, defeitos que já foram corrigidos, taxa de defeitos descobertos, entre outros. Antes de definir qual métrica utilizar, é importante considerar a complexidade do software testado, pois as métricas podem ser aplicadas nos diversos estágios do desenvolvimento do software, porém isso depende do foco da métrica, da maturidade do processo de testes da empresa e do tipo do projeto.

Se o objetivo for a melhoria do processo de testes para projetos futuros, a análise da métrica pode ser deixada para o fim do processo de desenvolvimento, utilizando a base histórica do projeto finalizado, mas se o objetivo for melhorar o projeto em andamento, a coleta de dados e a análise devem ser contínuas, podendo ser comparativas ou cumulativas.

As medidas de teste mais conhecidas incluem a cobertura e a qualidade:

- ✗ a cobertura é uma métrica quantitativa, utilizada para medir o quanto de código do software o teste conseguiu abranger. Não deve ser aplicada para aferir a qualidade dos testes realizados. Baseia-se na cobertura de requisitos ou pela cobertura de código executado e também pode demonstrar a quantidade de casos de teste executados *versus* a quantidade de casos de teste planejados;
- ✗ a qualidade tem por objetivo demonstrar o grau de confiabilidade, de estabilidade e de desempenho do software. Utiliza como base a avaliação dos resultados dos testes e das falhas identificadas durante o processo de teste.

Como complemento ilustrativo, veja algumas métricas conhecidas e utilizadas.

- ✖ **Métricas de estimativas de testes:** total de pontos de teste (PT), horas de teste primárias (HTP), índice de planejamento e controle (IPC), total de horas de teste (THT).
- ✖ **Métricas de teste de aceitação:** quantidade de testes de aceitação por funcionalidades, porcentagem de assertivas de teste de aceitação passando e falhando, funcionalidades testadas e entregues (*Running Tested Features* ou RTF).
- ✖ **Métricas para gestão de defeitos:** quantidade de defeitos encontrados, índice de densidade de defeitos, índice de severidade de defeitos, tempo para arrumar um defeito, tempo médio para encontrar um defeito, quantidade de falhas encontradas no produto, tipos de defeitos encontrados, densidade dos defeitos residuais, defeitos encontrados *versus* defeitos corrigidos
- ✖ **Métricas de gestão dos testes:** curva S, curva zero *bug bounce*, efetividade de caso de teste, efetividade e eficiência dos testes, tempo disponível para esforço de teste.

Síntese

Neste capítulo, tivemos uma visão abrangente sobre testes de software, uma atividade de apoio fundamental para garantir a qualidade de um software. Quando o software não é testado adequadamente, há uma grande probabilidade de falhas em seu uso que, dependendo do objetivo do software, pode acarretar em consequências muito sérias e grandes prejuízos. Vimos os aspectos principais do teste de software e a evolução dos objetivos que foram ganhando perspectivas diferenciadas com o passar dos anos. A apresentação de algumas definições de testes e princípios, da diferença conceitual entre erro ou *bug*, falha e defeitos. Discutimos os níveis ou fases dos testes: testes de unidade, de integração, de sistema e de aceitação. Abordamos as técnicas de testes funcionais e estruturais conhecidas popularmente como testes de caixa preta e de caixa branca, bem como os critérios de validação de cada uma delas. Por fim, foram citados alguns exemplos de métricas utilizadas nos processos de testes de software, tais como métrica de cobertura, de qualidade e pontos de testes.

Atividades

1. Indique se as afirmações são verdadeiras (V) ou falsas (F).
 - a) () O principal objetivo dos testes é mostrar a ausência de defeitos no software.
 - b) () Testes de sistema devem ser realizados após o software ser finalizado.
 - c) () Testes funcionais não podem ser automatizados.
 - d) () Os testes podem ser iniciados somente quando o software já estiver codificado.
 - e) () Os testes são muito fáceis de serem executados, basta utilizar o software sem a necessidade de aplicar alguma técnica ou planejamento.
2. Supondo um software em fase final de desenvolvimento e que já passou por testes funcionais e estruturais, qual teste poderia ainda ser realizado para que o software fosse liberado para os usuários finais?
3. A partir de qual nível de testes pode-se aplicar o teste caixa branca?
4. Indique qual técnica de teste avalia o comportamento externo do sistema sem se importar com a parte interna dele.

6

Manutenção de Software

No CICLO DE vida do produto de software, a entrega caracteriza o fim desse processo. É quando o software é instalado e aceito pelo cliente. No entanto, para que a aplicação continue sendo útil à organização, é inevitável a realização de contínuas mudanças, seja de caráter corretivo ou evolutivo (SOMMERVILLE, 2011).

DE FATO, AS organizações estão sujeitas há um processo de globalização e concorrência em que, para que se mantenham competitivas, devem se adequar às exigências mercadológicas. Ocorrem, assim, mudanças nas regras de negócio, nas estratégias organizacionais e inserção de novas tecnologias. Entender o que significa manutenção de software, e principalmente a abrangência do significado do termo, constitui um passo fundamental para o aprofundamento de soluções de problemas atrelados à atividade de desenvolvimento de software.

ESTE CAPÍTULO ABORDA os conceitos fundamentais sobre manutenção no ciclo de vida de software, bem como os tipos de manutenção e custos associados, procura entender o processo de evolução de software e apresentar técnicas e ferramentas de manutenção de software.

6.1 Fundamentação da manutenção de software

A atividade de manutenção de software é caracterizada pela modificação de um produto de software já entregue ao cliente, para correção de eventuais erros, melhora em seu desempenho ou qualquer outro atributo, ou ainda para adaptação desse produto a um ambiente modificado (IEEE, 2004).

A fase de manutenção é normalmente a mais longa do ciclo de vida do produto de software. O sistema é instalado e colocado em uso. Envolve atividades como a correção de erros que não foram descobertos nas fases anteriores de desenvolvimento, bem como a inserção de melhorias nas unidades do sistema e a extensão de novas funcionalidades em resposta às necessidades de negócio das organizações (SOMMERVILLE, 2011). Portanto, os produtos de software podem mudar devido a ações corretivas e não corretivas de software.

O SWEBOK apresenta algumas razões para a execução da manutenção (IEEE, 2004):

- × corrigir falhas;
- × melhorar o design;
- × implementar melhorias;
- × interface com outros softwares;
- × adaptar os programas de modo que a aplicação possa ser utilizada em diferentes hardwares, softwares, recursos do sistema e instalações de telecomunicações;
- × migrar o software legado;
- × desativar o software.

Nesse processo, existem cinco atividades principais do mantenedor:

- × manter o controle das funções do software no dia a dia;
- × manter o controle sobre as modificações do software;
- × aperfeiçoar as funções existentes;

- × identificar ameaças à segurança e corrigir vulnerabilidades de segurança;
- × prevenir a degradação da estrutura do software devido a constantes mudanças.

Apesar de sua importância, a fase de manutenção é conhecida como a mais cara e menos previsível do ciclo de vida, representando, em alguns casos, entre 67% e 90% dos custos totais (POLO et al, 1999). Portanto, são vários os desafios dessa fase e uma série de questões-chave deve ser tratada para garantir a manutenção eficaz de software.

Segundo a IEEE (2004), um dos desafios encontrados no processo de manutenção de software refere-se a problemas técnicos relacionados à compreensão limitada e ao tempo curto que é exigido da equipe de manutenção para realizar as modificações – principalmente, devido ao fato de que muitas vezes as modificações são realizadas por uma equipe diferente da que originou o software. Outro problema é a documentação deficiente. O reconhecimento de que o software deve ser documentado é um primeiro passo, mas a documentação deve ser comprehensível e consistente com o código-fonte para efetivamente auxiliar no processo. Frequentemente, é difícil ou impossível rastrear a evolução do software por meio de muitas versões ou lançamentos, devido às mudanças não estarem adequadamente documentadas (SOMMERVILLE, 2011).

Segundo Polo et al (1999), uma estrutura de trabalho adequada e com a identificação clara das tarefas que cada membro deve executar pode influenciar na produtividade. Os autores propõem a estrutura de equipe descrita a seguir.

- × Organização do cliente: conforme definido pela norma ISO/IEC 12207, corresponde ao adquirente, ou seja, à organização que possui o software e requer o serviço de manutenção.
- × Mantenedor: organização que fornece o serviço de manutenção.
- × Usuário: organização que utiliza o software mantido.

Essa estrutura de manutenção apresenta uma interligação entre os papéis e segue um fluxo de atividades. Por exemplo, a organização do cliente deve ter um responsável por enviar os pedidos à empresa que realiza as manutenções.

Do lado do mantenedor, o gerente de manutenção recebe os pedidos de manutenção e os analisa. Para os pedidos aceitos, é montado um cronograma. Em seguida, o chefe da manutenção prepara a fase da manutenção e também estabelece as normas e procedimentos que serão usados na manutenção. A equipe de manutenção implementa os pedidos de modificação conforme o cronograma. Por fim, o usuário faz uso do software comunicando os problemas ao *Help desk* (corresponde ao departamento de contato com os clientes), fechando o ciclo.

Outro ponto importante na fase de manutenção é fazer uma análise de impacto do software existente antes de realizar uma mudança e estabelecer algumas tarefas de análise de impacto (IEEE, 2004):

- × analisar as solicitações de mudanças;
- × replicar ou verificar o problema;
- × desenvolver opções para implementar a modificação;
- × elaborar documento de solicitação de mudança indicando as opções de execução;
- × obter a aprovação para opção de modificação selecionada.

A gravidade de um problema é muitas vezes usada para decidir como e quando este será corrigido. O engenheiro de software, em seguida, identifica os componentes afetados. Diversas soluções potenciais são fornecidas, seguidas por uma recomendação sobre o melhor curso de ação.

6.1.1 Evolução de software

As mudanças que ocorrerão em um software para deixá-lo mais completo, livre de erros ou adaptado ao seu ambiente podem ser definidas como atividades de evolução de software, conforme explica Sommerville (2011). A evolução de um sistema raramente pode ser considerada de maneira isolada. Alterações no ambiente levam a mudanças nos sistemas, que podem, então, provocar mais mudanças ambientais.

O fato de o sistema ser modificado em operação costuma aumentar as dificuldades e os custos com a evolução, bem como a complexidade de análise de impacto com outros sistemas.

Para obter melhor compreensão sobre a dinâmica da evolução dos programas, nas décadas de 1970 e 1980, foram realizados estudos para entender melhor as características da evolução de aplicações existentes. Após longos anos de estudos, foram definidas as “Leis de Lehman” relativas às mudanças, conforme mostra o quadro 6.1. Essas leis consideram não apenas o software em si, mas as características da organização que o desenvolve e o mantém.

Lehman e Belady defendem que essas leis são suscetíveis de serem verdadeiras para todos os tipos de sistemas de software de grandes organizações. Nesses sistemas, os requisitos estão mudando para refletir as necessidades de negócios. Novos releases do sistema são essenciais para fornecer valor ao negócio.

Quadro 6.1 – Leis de Lehman

Lei	Descrição
Mudança contínua	Um programa do mundo real deve necessariamente mudar ou se torna progressivamente menos útil nesse ambiente.
Aumento da complexidade	Como um programa em evolução, ele muda; sua estrutura tende a se tornar mais complexa. Recursos extras devem ser dedicados a preservar a simplicidade da estrutura.
Evolução de programas de grande porte	A evolução do programa é um processo de autorregulação. Atributos de sistema, como tamanho, tempo entre release e números de erros relatados, são aproximadamente inviáveis para cada release do sistema.
Estabilidade organizacional	Ao longo da vida de um programa, sua taxa de desenvolvimento é aproximadamente constante e independente dos recursos destinados ao desenvolvimento do sistema.
Conservação da familiaridade	Durante a vigência de um sistema, a mudança incremental em cada release é aproximadamente constante.

Lei	Descrição
Crescimento contínuo	A funcionalidade oferecida pelos sistemas tem de aumentar continuamente para manter a satisfação do usuário.
Declínio da qualidade	A qualidade dos sistemas cairá, a menos que eles sejam modificados para refletir mudanças em seu ambiente operacional.
Sistema de feedback	Os processos de evolução incorporam sistemas de feedback multiagentes, multi-loop, e deve-se retirá-los como sistemas de feedback para alcançar significativa melhoria do produto.

Fonte: Sommerville (2011).

6.1.2 Tipos de manutenção de software

A manutenção de software é o processo de alteração realizado depois que um programa é liberado para uso. Tais mudanças são implementadas por meio da modificação de componentes do sistema existente e, quando necessário, por meio do desenvolvimento de novos componentes (SOMMERVILLE, 2011).

Embora o processo de manutenção seja semelhante ao processo de desenvolvimento (mas não igual), pode envolver atividades de levantamento de requisitos, análise, projeto, implementação e testes, agora no contexto de um software existente. Essa semelhança pode ser maior ou menor, dependendo do tipo de manutenção a ser realizada.

A manutenção dos softwares pode ser classificada conforme sua necessidade. Para Sommerville (2011), a manutenção de software engloba as três atividades a seguir.

- ✗ **Correção de defeitos** (ou manutenção corretiva): consiste na correção de erros observados durante a fase de operação. Esse tipo de manutenção envolve retrabalho da equipe de desenvolvimento, gerando custos adicionais ao projeto. Erros relacionados à correção de código são relativamente barato. No entanto, a correção de erros

relacionados à má interpretação de requisitos é mais cara devido à demanda de mais tempo e recursos para o reprojeto do sistema.

- ✖ **Adaptação ambiental** (ou manutenção adaptativa): realiza alterações no software para que possa ser executado sobre o novo ambiente. Esse tipo de manutenção é necessária devido às necessidades de mudanças relacionadas a aspectos do ambiente do sistema, como o hardware, a plataforma do sistema operacional ou outro software de apoio que sofre uma mudança.
- ✖ **Adição de novas funcionalidades** (ou manutenção de aperfeiçoamento): é necessária quando os requisitos de sistema mudam em resposta às mudanças organizacionais ou de negócio. A escala de mudanças necessárias é relativamente maior do que os outros tipos de manutenção.

No SWEBOK é apontada uma quarta categoria, denominada **manutenção preventiva** (IEEE, 2004). Ela parte de uma observação reconhecida pelos mantenedores sobre o que poderá gerar algum tipo de erro no software; dessa forma, tal erro será tratado antes que um problema venha a ocorrer. A norma agrupa as categorias de manutenção como reativa e proativa, conforme o quadro 6.2.

Quadro 6.2 – Categorias de manutenção de software

	Correção	Aperfeiçoamento
Proativa	Preventiva	Aperfeiçoamento
Reativa	Corretiva	Adaptativa

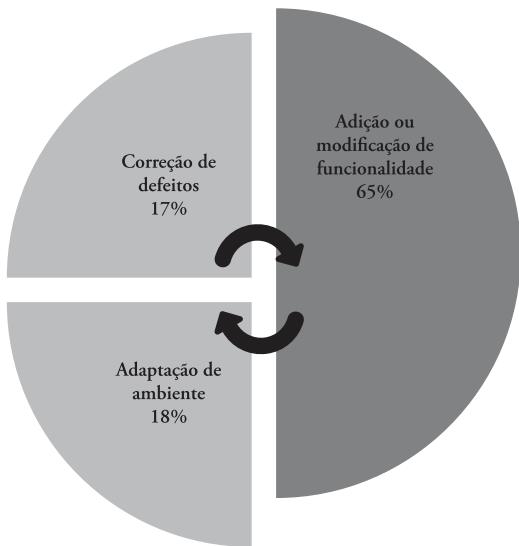
Fonte: IEEE (2004).

6.1.3 Custos de manutenção

Como já afirmado, as pesquisas em geral indicam que a fase de manutenção é considerada a mais cara do desenvolvimento de software e que os custos são maiores para implementar novas soluções do que para a correção de bugs identificados na fase de operação (SOMMERVILLE, 2011 apud KROGSTIE et al, 2005; LIENTZ et al, 1980).

A figura 6.1 indica que o percentual de custos com o desenvolvimento de novas funcionalidades ultrapassa 50% em relação aos demais tipos de manutenção.

Figura 6.1 – Distribuição do esforço de manutenção



Fonte: Sommerville (2011).

A manutenção do software envolve, normalmente, etapas de análise do sistema existente (entendimento do código e dos documentos associados), teste das mudanças e teste das partes já existentes, o que a torna uma etapa complexa e de alto custo.

Para tomar a decisão de continuar ou abandonar o projeto, devem-se considerar fatores como custo, confiabilidade do software após a manutenção, capacidade que possuirá de se adaptar a mudanças futuras, seu desempenho, limitações de suas funcionalidades atuais e opções propostas no mercado que atendem à necessidade da empresa de maneira mais completa, rápida e barata (PFLEGER, 2004).

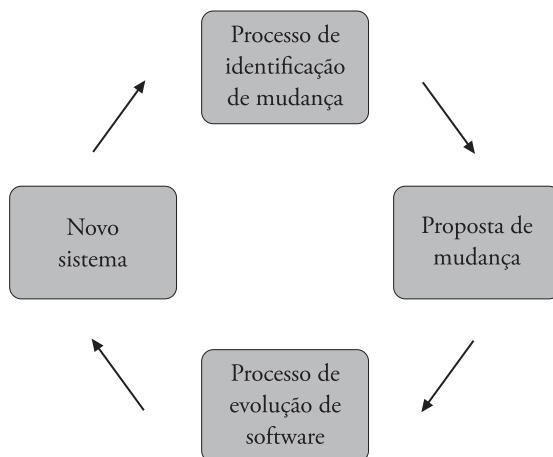
Por isso, vale a pena investir em um projeto de implementação de um sistema para redução de custos de mudanças futuras devido aos custos envolvidos na adição de novas funcionalidades após a liberação do software (SOM-

MERVILLE, 2011). Nesse sentido, o uso de boas técnicas de engenharia de software, como descrição precisa na fase de requisitos, desenvolvimento orientado a objetos, gerenciamento de configuração e mudanças, pode contribuir para a redução dos custos com manutenção. Além disso, considerando a atual situação industrial, foi criado, mais recentemente, o conceito de Reengenharia de Software, no qual, pelo uso das técnicas e ferramentas da Engenharia de Software, o software existente sofre uma “reforma geral”, cujo objetivo é aumentar a sua qualidade e atualizá-lo com respeito às novas tecnologias de interface e de hardware.

6.2 Processos de manutenção

Nas organizações, as propostas de mudanças no sistema são os acionadores para a evolução do software. As propostas de mudanças podem vir de requisitos já existentes que não tenham sido implementados no release do sistema, solicitações de novos requisitos, relatório de bugs do sistema apontados pelos usuários e novas ideias para melhoria do sistema (SOMMERVILLE, 2011). Os processos de mudanças são cílicos e continuam durante todo ciclo de vida de um sistema, conforme ilustra a figura 6.2.

Figura 6.2 – Processo de identificação de mudanças e evolução



Fonte: Sommerville (2011).

As propostas de mudança devem ser relacionadas aos componentes que precisam ser modificados. Conforme já comentado, realizar modificações em aplicações existentes demanda um esforço maior de análise e adaptações, além de maior atenção nas atividades de verificação, validação e integração com o sistema já em operação, a fim de evitar inserção de novos erros.

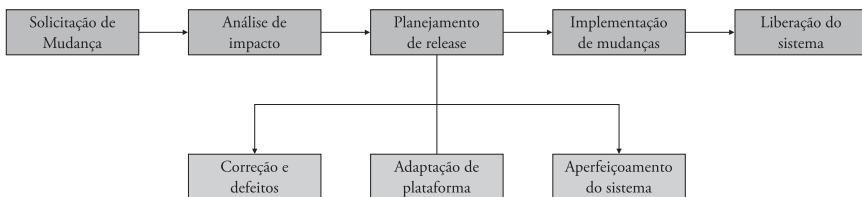
Esse processo não é trivial; mudanças ambientais costumam aumentar as dificuldades e aumentar os custos (SOMMERVILLE, 2011). Assim, a evolução de um sistema não pode ser considerada de modo isolado; é importante seguir um processo de manutenção que analise o impacto das mudanças.

Alguns modelos têm sido propostos para auxiliar as organizações nessa fase do ciclo de vida do produto de software. As próximas seções apresentam algumas dessas propostas.

6.2.1 Processo geral de evolução de software

A figura 6.3 mostra um processo geral de evolução de software adaptada por Sommerville (2011) apud Arthur (1998). O processo inclui atividades fundamentais de análise de impactos, planejamento do release, implementação de sistema e liberação do sistema para os clientes.

Figura 6.3 – Processo de evolução do sistema



Fonte: Sommerville (2011).

O custo e o impacto dessas mudanças são avaliados para ver quanto do sistema é afetado pelas mudanças e quanto poderia custar para implementá-las. Se as mudanças são aceitas, um novo release do sistema é planejado. No planejamento, é definido o tipo de manutenção (correção, adaptação ou

aperfeiçoamento). A decisão é tomada, e as mudanças são implementadas e validadas. Depois, uma nova versão do sistema é liberada.

Nesse processo, é importante a compreensão do programa existente, especialmente se a alteração não é feita por outra equipe de desenvolvimento. É necessário entender a estrutura do sistema e como a mudança pode afetá-lo. Para isso, é realizada uma análise detalhada dos requisitos, e possivelmente são necessárias mais discussões com os usuários.

A implementação de mudanças ocorre como uma iteração do desenvolvimento, em que as visões dos sistemas são projetadas, implementadas e testadas. A diferença é que no primeiro estágio da implementação deve ocorrer uma compreensão sobre a estrutura do programa original e como a mudança pode afetar as demais funcionalidades, especialmente se a equipe que vai realizar a manutenção é diferente da equipe que desenvolveu o sistema original.

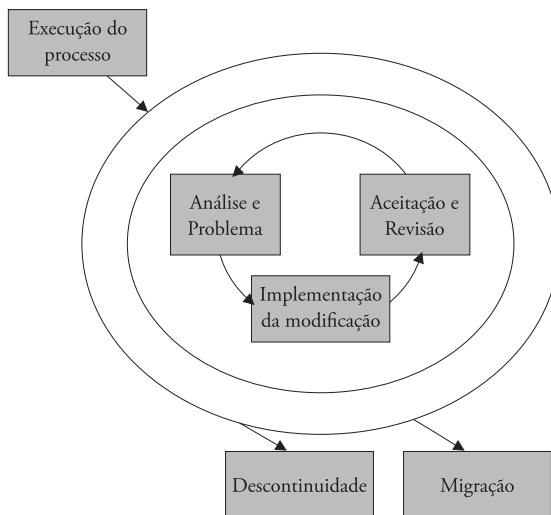
Sommerville (2011) explica que às vezes as mudanças são decorrentes de problemas que devem ser tratados com urgência e apresenta três motivos principais para essas urgências:

1. corrigir um defeito grave no sistema que compromete o andamento normal da aplicação;
2. defeitos oriundos de alterações realizadas no ambiente operacional que comprometem a continuidade do funcionamento do sistema;
3. implementação de mudanças inesperadas no funcionamento do negócio que executa o sistema, oriunda da concorrência ou de uma nova legislação que afete o sistema.

6.2.2 Processo de manutenção de software (Norma ISO/IEC 14.764)

A norma ISO/IEC 14.764 é um complemento do processo de manutenção da ISO/IEC 12207 e descreve com maior riqueza de detalhes as atividades necessárias para manter a integridade na realização da manutenção de software (ISO/IEC, 1999). A figura 6.4 ilustra o processo de manutenção da norma.

Figura 6.4 – Processo de manutenção de software da ISO/IEC 14.764



Fonte: IEEE (2004).

O processo de manutenção contém atividades e tarefas necessárias para modificar um produto de software existente, preservando a sua integridade. Essas atividades e tarefas são de responsabilidade do mantenedor e se resumem do modo descrito a seguir (ISO/IEC, 1999).

- Execução do processo:** fase inicial que consiste na definição de planos e procedimentos pelo mantenedor para que sejam executados durante a manutenção. Devem-se estabelecer procedimentos para solicitação, armazenamento, acompanhamento de pedidos de modificação e notificação a seus solicitantes. A fim de desenvolver uma estratégia e procedimentos adequados, a equipe de manutenção deve identificar os limites da manutenção, buscar alternativas para a análise desta, designar alguém para realizar a manutenção, verificar os recursos disponíveis, estimar os custos da manutenção, avaliar a manutenibilidade do sistema e determinar as exigências da transição.

Principais saídas dessa atividade: plano de manutenção; procedimentos de manutenção; procedimentos de resolução de proble-

mas; planos para feedback do usuário; plano de transição; plano de gerenciamento de configuração.

- b) **Análise do problema e da modificação:** antes de modificar um sistema, é necessário analisar os efeitos colaterais a que um software está sujeito após uma modificação – essa atividade visa à realização da análise dos impactos em decorrência da manutenção –, assim como analisar os impactos. Caberá ao mantenedor o desenvolvimento e a documentação de possíveis soluções. Durante a análise, deve-se levar em consideração o tipo da manutenção, os limites dela (recursos disponíveis, custos e tempo) e o seu desempenho (desempenho, segurança e integridade).

Principais saídas dessa atividade: análise de impacto (declaração do problema ou novo requisito, avaliação, classificação do tipo de manutenção, prioridade e estimativa inicial); opção recomendada; aprovação da modificação; documentação atualizada; estratégia de testes e requisitos atualizados.

- c) **Implementação da modificação:** execução das modificações especificadas e aprovadas nas fases anteriores. O mantenedor deverá seguir os mesmos procedimentos definidos no processo de desenvolvimento da ISO/IEC 12.207, em relação às técnicas de engenharia de software e atividades relacionadas.

Principais saídas desta atividade: planos e procedimentos de teste atualizados; documentação atualizada; código-fonte modificado; relatório de teste; métricas.

- d) **Revisão e aceitação da modificação:** garante que as modificações no sistema estão corretas e que foram realizadas em conformidade com as normas aprovadas, utilizando a metodologia correta. Durante a revisão, o mantenedor deve rastrear as exigências da solicitação de mudança desde o projeto ao código-fonte, verificar a funcionalidade do código, se o código está de acordo com os padrões da empresa, verificar os componentes de software utilizados, se os componentes novos foram integrados de maneira adequada, checar se a documentação foi atualizada e realizar os testes no software modificado.

Principais saídas dessa atividade: nova *baseline* com as modificações aceitas, as modificações rejeitadas, um relatório de aceitação da manutenção e o resultado dos testes realizados.

- e) **Migração:** durante a vida de um sistema, ele pode ter que ser modificado para funcionar em ambientes diferentes. Para migrar de um sistema para um novo ambiente, o mantenedor precisa determinar as ações necessárias para realizar a migração e, em seguida, desenvolver e documentar os passos necessários para efetuar a migração. O plano para a migração contém as etapas propostas pela ISO/IEC 12.207, na qual o mantenedor deve analisar as necessidades para realização da migração, notificando os usuários sobre a migração, fornecendo treinamento, proporcionando uma notificação de conclusão, avaliação do impacto do novo ambiente e arquivamento de dados.

Principais saídas dessa atividade: plano de migração, ferramentas de migração, aplicação migrada, notificação de conclusão e backup dos produtos e dados antigos.

- f) **Descontinuação do software:** caracteriza o final da vida útil de um software, em que as modificações necessárias não serão mais realizadas. Após a definição de descontinuidade do software, o mantenedor deverá desenvolver um plano de descontinuidade, notificando os usuários e os demais interessados a respeito da ação, além de arquivar seus dados relacionados.

Principais saídas dessa atividade: plano de descontinuidade; notificação aos usuários; usuários treinados; produto de software retirado e armazenamento de dados do software antigo.

Em ambos os modelos apresentados nesta seção, observa-se a sistemática em identificação das necessidades de manutenção, análise técnica e econômica. A diferença entre os modelos é que o modelo apresentado pela norma ISO/IEC 14.764 também fornece instruções sobre as atividades de migração do software e descontinuidade.

Não é suficiente simplesmente esperar que o aumento da qualidade resulte da manutenção de software. Mantenedores deve ter um programa de qualidade de software. A qualidade deve ser planejada e processos devem ser implemen-

tados para apoiar o processo de manutenção. As atividades e técnicas devem garantir a qualidade de software (SQA). O processo V & V (verificação e validação), ou seja, testes e auditorias, deve ser selecionado em conjunto com todos os outros processos para atingir o nível de qualidade desejado.

6.3 Técnicas para manutenção

Essa seção apresenta algumas das técnicas geralmente utilizadas na manutenção de software.

6.3.1 Reengenharia de software

Como discutido nas seções anteriores, é de extrema importância o entendimento sobre a aplicação a que se está realizando implementação de manutenções. Entretanto, muitos sistemas, especialmente os legados mais velhos, são difíceis de serem compreendidos e modificados (SOMMERVILLE, 2011). Isso é devido a fatores como:

- ✖ deterioração da estrutura do sistema em razão de uma série de mudanças;
- ✖ problemas de inteligibilidade devido a otimizações para conseguir desempenho;
- ✖ o sistema tem pouca documentação e ela não foi atualizada. O que quer dizer que a documentação descreve um estado anterior do sistema, mas não a configuração atual;
- ✖ as pessoas que criaram o sistema deixaram a empresa, ninguém pode explicar muitas decisões que foram tomadas.

De modo a superar esses problemas, pode-se utilizar a reengenharia de software, que pode ser de programas ou de dados, aplicada para:

1. redocumentação do sistema;
2. refatoração da arquitetura;
3. mudança de uma linguagem moderna;
4. modificações e atualizações da estrutura e dos dados do sistema.

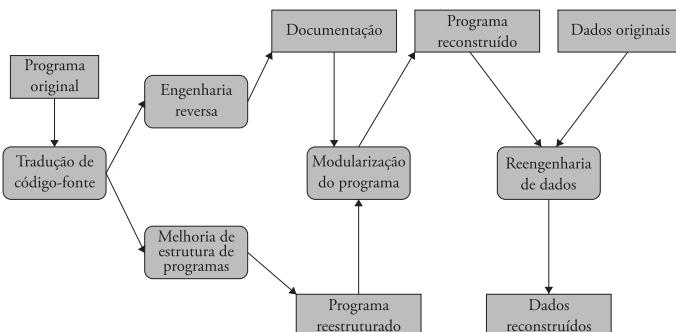
Segundo Pressman (2011), a reengenharia pode ocorrer em dois níveis de abstração: negócio e software. Em nível de negócio (*business process reengineering – BPR*), concentra-se nos processos para melhorar a competitividade em alguma área de negócio. No nível de software, a reengenharia examina os sistemas de informação e os aplicativos com a finalidade de reestruturá-los para que tenham melhor qualidade.

A reengenharia é considerada uma reconstrução de um sistema a partir de sua documentação e exame dele, sem alterar suas funcionalidades, apenas adicionando o que se julga necessário, no sentido de devolvê-lo ao usuário, com mais qualidade (PRESSMAN, 2011). No entanto, é uma atividade que absorve muitos recursos, por isso a organização necessita de uma estratégia prática. Portanto, é necessário evidenciar alguns aspectos, como: analisar o software e criar uma lista de critérios para que a reengenharia seja sistemática; analisar a estrutura do software antigo e avaliar se o tempo e os custos compensam.

A reengenharia de software é uma solução que deve ser considerada pelas empresas, pois pode diminuir os riscos de desenvolver uma nova aplicação com erros de especificação ou de implementação, que podem acarretar atrasos na entrega. Além disso, os custos para realizar a reengenharia podem ser显著mente menores do que desenvolver uma nova aplicação (SOMMERVILLE, 2011).

A seguir, a figura 6.5 representa todas as etapas da reengenharia, o entendimento do programa, a análise e o projeto, com o intuito de fornecer um nível de abstração maior que o do software antigo.

Figura 6.5 – Processo de reengenharia



Fonte: Sommerville (2011).

1. **Tradução de código-fonte:** a partir do uso de uma ferramenta de tradução, o programa é convertido de uma linguagem mais antiga para uma linguagem mais moderna da mesma linguagem ou outra diferente.
2. **Engenharia reversa:** o programa é analisado e as informações são extraídas a partir dele. O que a engenharia reversa faz é, justamente, partir de um baixo nível de abstração para um alto nível, pois, quando é iniciada a manutenção, tem-se apenas o sistema antigo, e então vai se criando alguma documentação até chegar à engenharia progressiva, para o desenvolvimento de um novo software.
3. **Melhoria de estrutura de programa:** a estrutura do programa é analisada e melhorada para que se torne mais fácil de entender.
4. **Modularização de programa:** partes relacionadas ao programa são agrupadas, e onde houver redundância, se apropriado, há remoção. Em alguns casos, pode haver refatoração – por exemplo, um sistema que usa vários repositórios de dados diferentes pode ser refeito para usar um único repositório. Esse processo é manual.
5. **Reengenharia de dados:** os dados processados pelo programa são alterados para acompanhar as mudanças do programa. Isso pode ser a redefinição dos esquemas de banco de dados e a conversão dos dados para nova estrutura.

A reengenharia de software abrange uma série de atividades que incluem análise do sistema; reestruturação de documentação; engenharia reversa; reestruturação de programas e dados, cuja finalidade é proporcionar às organizações um sistema de melhor qualidade e melhor manutenibilidade.

Os custos da reengenharia dependem da extensão do trabalho. A tradução do código é a opção mais barata, já a migração da arquitetura é a opção mais cara – por exemplo, converter um sistema escrito com uma arquitetura funcional para uma arquitetura orientada a objetos (SOMMERVILLE, 2011).

Uma maneira de determinar o custo-benefício da atividade de manutenção pode ser uma análise quantitativa a partir de uma avaliação do sistema atual e do custo associado ao suporte e à manutenção continuada de uma aplicação em comparação com os custos para desenvolver uma nova aplicação.

6.3.2 Gerenciamento de sistemas legados

Sistemas legados normalmente são sistemas críticos de negócio (SOM-MERVILLE, 2011). Portanto, normalmente precisam ser ampliados e adaptados às necessidades do negócio. Existem quatro opções estratégicas que podem ser tomadas pelas empresas:

- × descartar completamente o sistema – deve ser considerada quando a aplicação não contribui mais efetivamente para o negócio;
- × deixar o sistema inalterado e continuar com a manutenção regular – deve ser considerada quando o sistema ainda é necessário (o sistema é bastante estável e os usuários do sistema fazem poucas solicitações de mudanças);
- × reestruturar o sistema para melhorar sua manutenibilidade – deve ser escolhida quando a qualidade do sistema foi degradada pelas mudanças, e novas mudanças para o novo sistema ainda estão sendo propostas. Esse processo pode incluir o desenvolvimento de novos componentes de interface, para que o sistema original possa trabalhar com outros mais novos;
- × substituir a totalidade ou parte do sistema por um novo sistema – quando o sistema antigo não puder continuar em operação ou quando sistemas de prateleira podem permitir o desenvolvimento do novo sistema a um custo razoável. Em muitos casos, uma estratégia de substituição evolutiva pode ser adotada.

6.3.3 Engenharia reversa

A engenharia reversa é o processo de análise de software para identificar os componentes e suas inter-relações e criar representações do software sob outra forma ou em níveis mais altos de abstração (PRESSMAN, 2011), ou seja, consiste em analisar o sistema antigo com o objetivo de adaptá-lo à realidade atual, adicionando novas funcionalidades para atender aos usuários.

A diferença entre reengenharia de software e a engenharia reversa é que a segunda é um subconjunto da primeira, visto que ela é feita a partir do exame minucioso do código do produto e será de grande valia no contexto total da reengenharia, que é considerada mais abrangente.

A engenharia reversa é um processo de recuperação do projeto e visa o entendimento geral por meio da documentação gerada. De maneira a facilitar as manutenções futuras. As ferramentas de engenharia reversa extraem informações do projeto de dados, arquitetura e procedural da aplicação existente.

Pressman (2011) aponta três usos da engenharia reversa: entender os dados, entender o processamento e entender a interface. A engenharia reversa, para entender os dados, ocorre em diferentes níveis de abstração e é a primeira tarefa da reengenharia. O primeiro passo é o entendimento da estrutura interna do código. Em seguida, em nível de sistema é entendida a estrutura global dos dados e definida uma nova estrutura para todo o sistema.

- ✖ **Estruturas internas:** as técnicas de engenharia reserva focalizam a definição das classes de objetos. É feito um exame do código do programa para agrupar variáveis relacionadas, ou seja, busca agrupar estruturas de dados comuns do programa, como estruturas de registros, arquivos de listas e outros.
- ✖ **Estrutura de base de dados:** entendimento da estrutura lógica e física do programa, ou seja, busca o entendimento dos objetos e suas relações.

A engenharia reversa, para entender o processamento, é realizada a partir do entendimento das abstrações procedurais representadas pelo código fonte. Isso é feito por meio da análise das várias abstrações do sistema, como: sistema, programa, componente, padrão e instruções.

Já para entender uma interface, por meio da engenharia reversa, a estrutura e o comportamento dessa interface deve ser especificada. Três perguntas podem ser respondidas para guiar essa especificação: (1) Quais são as ações básicas que a interface deve processar? (2) Qual a descrição compacta da resposta comportamental do sistema a essas ações? (3) Que conceito de equivalência de interfaces é relevante aqui?

6.3.4 Refatoração

As mudanças inseridas no código, com o passar do tempo, farão com que o projeto do programa sofra um deterioramento. A refatoração é um

processo de fazer melhorias em um programa para diminuir a degradação gradual resultante das mudanças (SOMMERVILLE, 2011).

Embora a reengenharia de software e a refatoração visem a tornar o software mais fácil de ser entendido e de se realizarem modificações, são processos distintos, conforme Sommerville (2011):

- × **reengenharia de software** – ocorre após algum tempo de manutenções realizadas e com o aumento dos custos de manutenção. A partir do uso de ferramentas automatizadas é possível processar e reestruturar um sistema legado para criar um novo sistema mais fácil de realizar manutenções.
- × **refatoração** – processo contínuo de melhoria ao longo do processo de desenvolvimento e evolução, com o intuito de evitar a degradação do código, que aumenta os custos e as dificuldades de manutenção de um sistema. Isso significa melhorar um programa para modificar sua estrutura a fim de reduzir sua complexidade ou para torná-lo mais compreensível. Portanto, a refatoração pode ser considerada uma manutenção preventiva que reduz os problemas de mudanças no futuro.

Fowler et al. (1999) apresentam situações que denominam “maus cheiros”, em que o uso da refatoração pode auxiliar na manutenção de software.

- × **Código duplicado:** são os códigos duplicados ou semelhantes utilizados em diferentes partes do programa. Implementado como um único método e utilizado quando necessário.
- × **Métodos longos:** são métodos longos, portanto, deve-se reprojetar como uma série de métodos mais curtos.
- × **Declarações switch (case):** envolvem duplicações em situações em que o switch depende do tipo de algum valor. As declarações podem ser espalhadas em torno do programa. Na orientação a objetos, isso é evitado a partir do polimorfismo.
- × **Aglutinações dos dados:** ocorre quando um mesmo grupo de itens de dados (classes, parâmetros em métodos) reincide em vários

lugares de um programa. Podem ser substituídos por objetos que encapsulem todos os dados.

- ✖ **Generalidade especulativa:** ocorre quando os desenvolvedores incluem generalidades em um programa, pois estas podem ser necessárias no futuro e muitas vezes podem ser removidas.

A refatoração pode arrumar o código, pois a refatoração sistemática o ajuda a conservar sua forma. Quando a refatoração do código é feita corretamente, o projeto fica melhor, mais legível e menos propenso a falhas.

6.4 Ferramentas de manutenção de software

Este tópico abrange ferramentas que são particularmente importantes na manutenção de software, em que o software existente está sendo modificado. Exemplos sobre o programa de compreensão incluem (IEEE, 2004):

- ✖ máquinas de corte de programas, que selecionar apenas partes de um programa afetado por uma mudança;
- ✖ analisadores estáticos, que permitem a visualização geral e resumos de conteúdo do programa;
- ✖ analisadores dinâmicos, que permitem que o mantenedor trace o caminho de execução de um programa;
- ✖ analisadores de fluxo de dados, que permitem que o mantenedor rastreie todos os fluxos de um programa de dados possíveis;
- ✖ referências cruzadas, que geram índices de componentes do programa;
- ✖ analisadores de dependência, que ajudam mantenedores a analisar e compreender as relações entre componentes de um programa.

As ferramentas de engenharia reversa auxiliam no processo, trabalhando com um produto já existente para criar artefatos como descrições de especificação e design, que podem então ser transformados para gerar um novo produto a partir de um antigo. Mantenedores também usam teste de software,

gerenciamento de configuração de software, documentação de software e ferramentas de medição de software.

Síntese

Este capítulo apresentou a importância da manutenção para a continuidade do uso de sistemas de informação. A manutenção é uma atividade que vai ser realizada por todo ciclo de vida do produto de software e que pode ser feita tanto para correção de defeitos como para inserção e/ou melhorias da aplicação existente.

Também foram apresentados dois processos de manutenção e técnicas de manutenção, como reengenharia de software, engenharia reversa e fatoração de software, que auxiliam na análise de sistemas cujo objetivo é o entendimento da aplicação e reestruturação de projeto, de modo a melhorar sua estrutura e facilitar as manutenções futuras.

Atividades

1. Explique por que um sistema de software usado em um ambiente real deve mudar ou torna-se progressivamente menos útil.
2. Quais as diferenças entre os quatro tipos de manutenção? Explique-as.
3. O que é reengenharia de software e que benefícios pode-se ter com sua adoção?
4. Julgue as afirmativas em verdadeiras (V) ou falsas (F).
 - () Entre outras coisas, manutenção significa implantar novas funções.
 - () Padronizar comentários em um programa dificulta a legibilidade e o entendimento.
 - () Um módulo altamente coeso é também um módulo fracamente acoplado.
 - () Os custos de manutenção tendem a decrescer com os sistemas legados.

- () Normalmente o processo de manutenção é encarado pela equipe como uma atividade nobre, dentro do ciclo de vida do sistema.
- () Coesão e acoplamento são critérios de modificabilidade de um software.
- () Entre outras coisas, manutenção significa corrigir erros.
- () Migração é uma subárea da Engenharia Reversa.
- () Reestruturação de dados tem relação com padronização de dados.
- () Gerenciamento de configuração permite controlar as mudanças nos equipamentos do usuário.

7

Gerenciamento de Configuração de Software

No DESENVOLVIMENTO DE software, as mudanças são constantes e muitas vezes inevitáveis. Tal fato ocorre devido ao entendimento do usuário, que muda conforme a necessidade do negócio, e ao ambiente, que também muda constantemente, além da legislação que está em constante atualização. Com tantas mudanças, é necessária alguma maneira de organização.

A ADOÇÃO DO gerenciamento de configuração de software no desenvolvimento de sistemas direciona como estruturar, gerenciar e controlar o software por todo seu ciclo de vida, proporcionando acompanhamento de todo o processo. O objetivo deste capítulo é apresentar os principais conceitos de gerenciamento de configuração de software e fornecer base para a compreensão das atividades envolvidas nesse processo.

7.1 Fundamentação do gerenciamento de configurações

No ambiente de desenvolvimento de software, a gestão de configuração tem caráter essencial quando as modificações e as alterações são inevitáveis durante seu ciclo de vida. Essas alterações podem perturbar o ambiente de desenvolvimento, principalmente quando não são analisadas antes de serem realizadas, não são registradas antes de serem codificadas e não são relatadas aos envolvidos.

Em sistemas nos quais a confiabilidade é fator de sucesso na utilização, a falta de controle nas alterações pode trazer transtornos no cumprimento de objetivos e prejuízos aos negócios. Além disso, atualmente o desperdício de tempo e dinheiro não é aceitável.

Partindo do princípio de que um sistema pode ser definido como uma combinação de elementos organizados para atingir um ou mais objetivos, a configuração trata das características funcionais e físicas (software e hardware), estabelecidas na documentação técnica construída (INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, 2010). A configuração de um sistema também pode ser considerada como uma coleção de versões específicas de itens de software ou de hardware combinados de acordo com procedimentos de compilação específicos para atender a uma finalidade.

Nesse contexto, o termo *gerenciamento de configuração de software* é formalmente definido pelos autores da engenharia de software como

a disciplina de identificar a configuração de um sistema em pontos distintos no tempo, com a finalidade de controlar sistematicamente as mudanças na configuração e manter a integridade e rastreabilidade da configuração, ao longo do ciclo de vida do software (INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, 2010).

o desenvolvimento e aplicação de padrões e procedimentos para gerenciar um produto de sistema em desenvolvimento (SOMMERVILLE, 2011, p. 457).

uma atividade guarda-chuva que é aplicada ao longo de todo o processo de software. (PRESSMAN, 2011, p. 220).

De acordo com essas definições, o gerenciamento de configuração de software é um processo de ciclo de vida do software que beneficia a guarda-

-chuva gestão do projeto, as atividades de desenvolvimento e de manutenção, as atividades de garantia de qualidade e até mesmo os clientes e usuários do produto final. O gerenciamento de configuração de software está intimamente relacionado com a atividade de garantia de qualidade de software, a qual garante que os produtos e os processos de software, no ciclo de vida do projeto, estejam em conformidade com os requisitos especificados, fornecendo a confiança de que a qualidade está incorporada ao software.

As atividades de configuração de software basicamente são planejamento do gerenciamento de configuração de software, identificação de configuração de software, controle de configuração de software, registro de status de configuração de software, auditoria de configuração de software e gerenciamento e entrega do software.

7.2 Planejamento do gerenciamento de configuração de software

Para a realização do planejamento do gerenciamento de configuração de software, é necessário entender o contexto organizacional e as relações entre os elementos organizacionais, visto que interage com várias outras atividades ou elementos organizacionais.

Os elementos organizacionais responsáveis pelos processos de suporte à Engenharia de Software podem ser compreendidos de várias maneiras:

- ✖ o software é frequentemente desenvolvido como parte de um sistema maior, contendo outros elementos, como hardwares. Nesse caso, as atividades do gerenciamento de configuração de software ocorrem paralelamente ao gerenciamento do hardware;
- ✖ as políticas e as restrições organizacionais podem interferir e até mesmo orientar o gerenciamento de configuração de software; os procedimentos estabelecidos em níveis corporativos ou outros níveis organizacionais podem influenciar ou prescrever a concepção e a implementação do processo de gerenciamento de configuração de software para determinado projeto, por exemplo, o contrato entre o contratante e contratado/fornecedor pode conter cláusulas que afetem o processo de compra ou a prestação de serviço, visto

que em alguns momentos certas auditorias de configuração podem ser necessárias;

- × o gerenciamento de configuração de software também pode interagir com a atividade de garantia de qualidade de uma organização, por exemplo, em questões como o gerenciamento de registros de itens de não conformidade organizacional; gerenciar itens de não conformidade geralmente é responsabilidade da atividade de garantia de qualidade, no entanto, o gerenciamento de configuração de software pode auxiliar no rastreamento e no relato de itens de configuração de software pertencentes a essa categoria.

De maneira geral, o planejamento de um processo de gerenciamento de configuração de software deve ser aderente ao contexto organizacional, sempre acompanhando as políticas de restrições, as orientações institucionais, a natureza do projeto (por exemplo, tamanho, criticidade e segurança), as responsabilidades organizacionais, os recursos disponíveis, o processo de seleção e implementação de ferramentas, além da gestão do controle de contratos e de fornecedores (INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, 2012).

Os resultados da atividade de planejamento são registrados em um plano de gerenciamento de configuração de software que normalmente está sujeito a revisão e auditoria de qualidade. Como o plano deve observar vários elementos organizacionais, e para evitar confusão sobre quem executará determinadas atividades ou tarefas do gerenciamento de configuração de software, os papéis/responsáveis a serem envolvidos no processo precisam ser claramente identificados.

As responsabilidades específicas para determinadas atividades ou tarefas dentro do gerenciamento de configuração de software precisam ser atribuídas, bem como a autoridade geral e os canais de comunicação devem ser identificados, embora isso também possa ser realizado na fase de planejamento de gerenciamento de projetos ou de garantia de qualidade. Normalmente, a prática do gerenciamento de configuração de software requer um conjunto de ferramentas de apoio. Como para qualquer área de Engenharia de Software, a seleção e a implementação de ferramentas devem ser cuidadosamente planejadas e, para tanto, as seguintes perguntas devem ser consideradas (INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, 2010):

- ✖ o que motiva a aquisição das ferramentas, a partir de uma perspectiva organizacional?
- ✖ podemos usar ferramentas comerciais ou desenvolvê-las?
- ✖ quais são as restrições impostas pela organização e seu contexto técnico?
- ✖ como os projetos usarão as ferramentas?
- ✖ quem pagará pela aquisição, pela manutenção, pelo treinamento e pela customização das ferramentas?
- ✖ como as ferramentas serão implantadas, por exemplo, para todos os projetos ou apenas em projetos específicos?
- ✖ quem é responsável pela introdução das ferramentas?
- ✖ qual é o plano para o uso das ferramentas no futuro?
- ✖ quão adaptáveis são as ferramentas?
- ✖ a capacidade das ferramentas é compatível com as estratégias organizacionais planejadas?
- ✖ as ferramentas se integram entre si? Ou com outras ferramentas em uso na organização?
- ✖ o repositório mantido pela ferramenta de controle de versão pode ser portado para outra ferramenta de controle de versão enquanto mantém o histórico completo dos itens de configuração?

Um projeto de software pode adquirir ou fazer uso de produtos de software comprados, como controladores de versionamento ou outras ferramentas, dessa maneira, o planejamento do gerenciamento de configuração de software deve considerar como esses itens serão incorporados (integrados nas bibliotecas do projeto) e como as alterações ou atualizações serão avaliadas e gerenciadas.

Quando um item de software, a ser controlado, interage com outro software ou item de hardware, uma alteração em qualquer item pode afetar o outro; assim, o planejamento para o processo do gerenciamento de configuração de software deve considerar como os itens de interface serão

identificados e como as alterações nos itens serão gerenciadas e comunicadas. Após a implantação do processo de gerenciamento de configuração de software, pode ser necessário algum grau de monitoramento para assegurar que as definições do plano sejam devidamente aplicadas, e normalmente existirão requisitos específicos para garantir a conformidade com processos e procedimentos específicos.

O uso de ferramentas de gerenciamento de configuração de software integradas com capacidade de controle de processo pode facilitar a tarefa de monitoramento. Algumas ferramentas facilitam a conformidade do processo, ao mesmo tempo que fornecem flexibilidade ao engenheiro de software para adaptar os procedimentos. Outras ferramentas reforçam o processo, deixando o engenheiro de software com menos flexibilidade. As exigências de monitoramento e o nível de flexibilidade a serem fornecidos ao engenheiro de software são considerações importantes na seleção das ferramentas.

Para garantir que o gerenciamento de configuração de software está sendo atendido conforme planejado, as medições podem ser projetadas para fornecer *insights* sobre o funcionamento do processo do gerenciamento de configuração de software. As medições são úteis na caracterização do estado atual do processo, bem como no fornecimento de uma base para fazer comparações ao longo do tempo. A análise das medições pode provocar mudanças de processo e atualizações do gerenciamento de configuração de software.

7.3 Identificação de configuração de software

Nessa atividade, ocorre a identificação dos itens a serem controlados, além do estabelecimento dos esquemas de identificação dos itens e suas versões, bem como a definição das ferramentas e das técnicas a serem utilizadas na aquisição e na administração desses itens (INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, 2012).

Entendemos anteriormente que a configuração de software trata as características funcionais e físicas do hardware ou do software, já um item de configuração de software é um elemento ou uma agregação de hardware ou de software, ou ambos, projetado para ser gerenciado como um único elemento. Por exemplo, um item de configuração de software pode ser um plano, uma

especificação e documentação de projeto, u material de teste (plano de testes, evidências...), um código-fonte, uma biblioteca de código ou componentes, um dicionário de dados, uma documentação para instalação, entre outros.

A identificação dos itens a serem controlados não é uma tarefa simples, necessita da compreensão da configuração do software no contexto da configuração do sistema, da compreensão da seleção de itens de configuração de software, do desenvolvimento de uma estratégia para “rotular” os itens de software, da descrição de suas relações e da identificação do procedimento para sua aquisição (HASS, 2013). As relações estruturais entre os itens de configuração de software selecionados e suas partes constituintes afetam outras atividades ou tarefas do gerenciamento de configuração de software, como a criação de software ou a análise do impacto das mudanças propostas. O rastreamento adequado dessas relações também é importante para apoiar a rastreabilidade.

A concepção do esquema de identificação para os itens de configuração de software deve considerar a necessidade de mapear itens identificados para a estrutura de software, bem como a necessidade de apoiar a evolução dos itens de software e suas relações. Na sequência, entenderemos alguns termos que cercam o gerenciamento de configuração, mais especificamente a identificação de configuração de software.

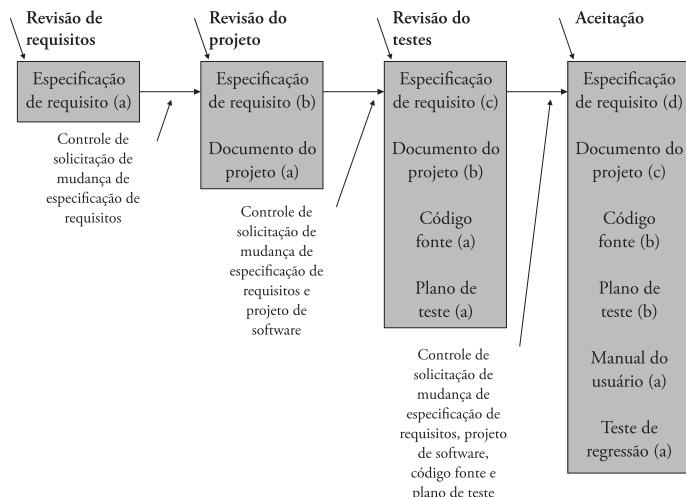
- ✖ **Versão do software:** os itens de software evoluem à medida que um projeto de software prossegue. Uma versão de um item de software é uma instância identificada de um item e pode ser pensado como um estado de um item em evolução;

Baseline: uma linha de base ou uma linha de referência de software é uma versão formalmente aprovada de um item de configuração, designada e fixa em um momento específico durante o ciclo de vida do item de configuração. O termo também é usado para se referir a uma versão específica de um item de configuração de software que foi acordado. Em ambos os casos, a *baseline* só pode ser alterada por procedimentos formais de controle de mudança; além disso, com todas as alterações aprovadas, representa a configuração aprovada atual. E *baselines* comumente usadas incluem *baselines* funcionais, alocadas, de desenvolvimento e de produtos:

- ✗ *baseline* funcional corresponde aos requisitos analisados do sistema;
- ✗ *baseline* alocada corresponde à especificação de requisitos de software revistos e à especificação de requisitos de interface de software;
- ✗ *baseline* de desenvolvimento representa a evolução da configuração do software em momentos selecionados durante o ciclo de vida do software;
- ✗ *baseline* do produto corresponde ao produto de software concluído para a integração do sistema; as *baselines* a serem usadas para determinado projeto, junto aos níveis de autoridade associados necessários para a aprovação da mudança, são tipicamente identificadas no plano de gerenciamento de configuração de software.
- ✗ **Aquisição de itens de configuração de software:** os itens de configuração de software são colocados sob o controle do gerenciamento de configuração de software em momentos diferentes. Ou seja, eles são incorporados em uma *baseline* específica em determinado ponto no ciclo de vida do software. O evento desencadeante é a conclusão de alguma tarefa de aceitação formal, como uma revisão formal.

A figura 7.1 representa o crescimento dos itens de *baseline* à medida que o ciclo de vida prossegue. Os índices utilizados na figura indicam versões dos itens em evolução:

- ✗ na aquisição de itens de configuração de software, sua origem e integridade inicial devem ser estabelecidas;
- ✗ após a aquisição de itens de configuração de software, as alterações do item devem ser formalmente aprovadas conforme apropriado para os itens de configuração de software e a *baseline* envolvida, como definido no plano de gerenciamento de configuração de software;
- ✗ após a aprovação, o item é incorporado à *baseline* do software de acordo com o procedimento apropriado.

Figura 7.1 - Exemplo de crescimento dos itens de *baseline*

Fonte: Adaptado de International Organization for Standardization (2010).

- ✗ **A biblioteca de software:** uma biblioteca de software é uma coleção controlada de software e documentação relacionada projetada para auxiliar no desenvolvimento, no uso ou na manutenção de software e também é instrumental na gestão de lançamento de software e atividades de entrega. Vários tipos de bibliotecas podem ser usados, cada um correspondendo ao nível particular de maturidade do software. Por exemplo, uma biblioteca de trabalho pode suportar codificação e uma biblioteca de suporte de projeto pode suportar testes, enquanto uma biblioteca principal pode ser usada para produtos concluídos.

Um nível apropriado de controle de gerenciamento de configuração de software está associado a cada biblioteca. A segurança, em termos de controle de acesso, e as facilidades de backup são um aspecto-chave do gerenciamento de bibliotecas. Nesse ponto, a ferramenta utilizada para cada biblioteca deve suportar as necessidades de controle do gerenciamento de configuração de software para aquela biblioteca, tanto em termos de controle de itens de configuração de software como de controle de acesso à biblioteca.

No nível de biblioteca de trabalho, esse é um recurso de gerenciamento de código que atende a desenvolvedores, pois é focado em gerenciar as versões de itens de software, apoiando as atividades de vários desenvolvedores. Em níveis mais altos de controle, o acesso é mais restrito e o gerenciamento de configuração de software é o usuário principal. Essas bibliotecas são também uma importante fonte de informação para medições de progresso do trabalho.

7.4 Controle de configuração do software

O controle de configuração está relacionado ao gerenciamento de alterações durante o ciclo de vida do software, ou seja, abrange o processo de determinação das alterações a serem realizadas, o nível de autoridade para aprovar as alterações e o apoio necessário para a realização dessas alterações.

7.4.1 Solicitação, avaliação e aprovação das alterações

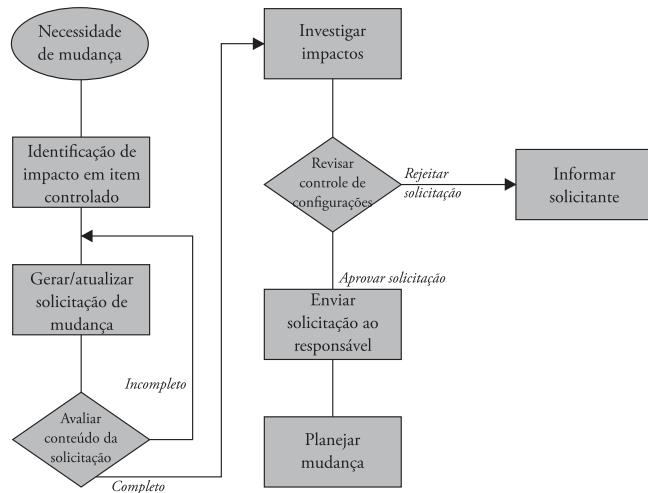
O primeiro passo no gerenciamento de mudanças é determinar quais mudanças devem ser feitas. O processo de solicitação de mudança de software, ilustrado na figura 7.2, apresenta uma proposta dos procedimentos formais para submeter e registrar solicitações de mudança, avaliar o custo e o impacto de uma alteração proposta e aceitar, modificar, adiar ou rejeitar a mudança solicitada. Uma solicitação de mudança pode ser uma solicitação para aumentar ou diminuir o escopo do projeto; modificar políticas, processos, planos ou procedimentos do projeto; modificar custos ou orçamento do projeto.

As solicitações de mudança podem ser originadas por qualquer pessoa em qualquer ponto do ciclo de vida do software, podendo incluir uma solução sugerida e uma prioridade solicitada. Independentemente da origem da solicitação de mudança, o tipo de alteração necessária é normalmente gravado em uma ferramenta de controle da solicitação de mudança. Isso provê uma oportunidade para monitorar defeitos e coletar medições de atividade de mudança por tipo de alteração.

Uma vez que uma solicitação de mudança é recebida, uma avaliação técnica, normalmente conhecida como análise de impacto, é realizada para determinar a extensão das modificações que serão necessárias para atender à solicitação. Por fim, uma avaliação dos aspectos técnicos e gerenciais do pedido de

mudança deve ser feito para a emitir o resultado da solicitação, ou seja, se será aceita, rejeitada ou ainda se há necessidade de alteração na solicitação.

Figura 7.2 - Processo de controle de mudanças



Fonte: Adaptada de International Organization for Standardization (2010).

Normalmente, a solicitação de mudança é autorizada por um comitê de controle de configurações, no entanto, em projetos menores, um indivíduo pode ser designado para o papel de autorizador ao invés de um conjunto de pessoas e pode haver vários níveis de autorizadores de mudança dependendo de uma variedade de critérios, como a criticidade do item envolvido, a natureza da mudança, o impacto no orçamento e no cronograma.

Como mencionado, um processo efetivo de solicitação de mudança de software requer o uso de ferramentas e procedimentos de suporte para registrar as solicitações de mudanças, reforçar o fluxo do processo de mudança de maneira automatizada, capturar decisões do comitê e relatar informações sobre processos de mudança.

7.4.2 Implementando as alterações de software

As solicitações de alterações de software são implementadas com os procedimentos de software definidos de acordo com o planejamento realizado,

uma vez que várias solicitações aprovadas podem ser implementadas simultaneamente e é necessário fornecer um meio para rastrear quais alterações são incorporadas em quais versões do software. Como parte do encerramento do processo de alteração, as alterações concluídas podem ser submetidas a auditorias de configuração e verificação da qualidade do software, o que inclui garantir que somente as alterações aprovadas tenham sido realizadas.

O processo de solicitação de mudança de software normalmente documenta as informações de aprovação do gerenciamento de configuração de software para a alteração. Essas alterações podem ser suportadas por ferramentas de controle de versão de código-fonte, que permitem que os engenheiros de software acompanhem e documentem mudanças no código-fonte. Tais ferramentas fornecem um repositório único para armazenar o código-fonte, podendo impedir que mais de um engenheiro de software edite o mesmo módulo ao mesmo tempo, além de registrar todas as alterações realizadas.

Os engenheiros de software verificam os módulos fora do repositório, fazem alterações, documentam as alterações e, em seguida, salvam os módulos editados no repositório. Se necessário, as alterações também podem ser descartadas, restaurando uma *baseline* anterior. Algumas ferramentas mais robustas podem suportar ambientes de desenvolvimento paralelo e geograficamente distribuídos.

7.5 Registro de status da configuração

O registro de status da configuração de software é a atividade de gerenciamento de configuração que consiste na coleta, no armazenamento e na criação das informações necessárias para gerenciar uma configuração de maneira eficaz. Como em qualquer sistema de informação, as informações de status de configuração a serem gerenciadas para as configurações em evolução devem ser identificadas, coletadas e mantidas.

Normalmente, várias informações e medições são necessárias para suportar o processo de gerenciamento de configuração de software. Os tipos de informação incluem a identificação da configuração aprovada, a identificação e o estado atual da implementação das alterações e os desvios ocorridos no processo. As informações geradas podem ser utilizadas por vários par-

ticipantes do projeto, incluindo a equipe de desenvolvimento, a equipe de manutenção, o gerente do projeto e a equipe de auditoria e qualidade. Tais informações podem ser consultadas *ad hoc* (quando necessário) para responder a perguntas específicas ou ainda possuir uma produção periódica preestabelecida em formato de relatório.

Algumas informações produzidas pela atividade de registro de status da configuração de software durante o curso do ciclo de vida podem se tornar registros de garantia de qualidade. Além de relatar o status atual da configuração, as informações obtidas podem servir como base para várias medições, como o número de solicitações de alterações ou o tempo médio necessário para implementar uma solicitação de alteração.

7.6 Auditoria da configuração

Segundo o Institute of Electrical and Electronics Engineers (2012), uma auditoria de software é um exame independente de um produto de trabalho ou conjunto de produtos de trabalho para avaliar o cumprimento de especificações, padrões, acordos contratuais ou outros critérios. As auditorias são conduzidas de acordo com um processo bem definido que consiste em vários papéis e responsabilidades do auditor, exigindo um número de indivíduos para executar uma variedade de tarefas ao longo de um período de tempo bastante curto. Consequentemente, cada auditoria deve ser cuidadosamente planejada.

A auditoria de configuração do software determina a extensão em que um item satisfaz as características funcionais e físicas necessárias. As auditorias informais podem ser realizadas em pontos-chave do ciclo de vida e dois tipos de auditorias formais podem ser normalmente encontradas na indústria de software:

- ✖ **auditoria de configuração funcional do software** – o objetivo dessa auditoria é garantir que o item de software auditado seja consistente com suas especificações de funcionamento;
- ✖ **auditoria de configuração física de software** – o objetivo dessa auditoria é garantir que a documentação do projeto seja consistente com o produto de software construído.

As auditorias podem ser realizadas durante o processo de desenvolvimento para investigar o status atual de elementos específicos da configuração. Nesse caso, uma auditoria poderia ser aplicada a itens de *baseline* amostral, para garantir que o desempenho é consistente com as especificações ou para garantir que a evolução da documentação continua a ser consistente com o item de *baseline* em desenvolvimento.

7.7 Gerenciamento e entrega do release de software

Release refere-se à distribuição de um item de configuração de software, o que inclui lançamentos internos, bem como a distribuição aos clientes. Quando múltiplas versões de um item de software estão disponíveis para entrega, como versões para plataformas variadas ou versões com diferentes capacidades, frequentemente é necessário recriar versões específicas e empacotar os materiais corretos para a entrega da versão. A biblioteca de software é um elemento-chave na realização da atividade de entrega do *release* de software.

A construção e a manutenção de software exigem a definição das versões corretas de itens de configuração de software, usando os dados de configuração apropriados em um programa executável para a entrega a um cliente ou a outro destinatário, como a atividade de teste. Além das novas versões do software, também é necessário que o gerenciamento de configuração de software tenha a capacidade de reproduzir versões anteriores para fins de recuperação, teste, manutenção ou liberação adicional, podendo ser necessário recriar uma cópia exata de um item de configuração de software. Nesse caso, as ferramentas de suporte e as instruções de compilação associadas precisam estar sob controle do gerenciamento de configuração de software, para garantir a disponibilidade das versões corretas das ferramentas.

O uso de ferramenta de entrega do *release* é útil para selecionar as versões corretas de itens de software a determinado ambiente de destino e para automatizar o processo de construção do software a partir das versões selecionadas e dos dados de configuração apropriados. A gestão da entrega de versões do software abrange a identificação, o empacotamento e a liberação dos elemen-

tos de um produto, por exemplo, um programa executável, a documentação e os dados de configuração.

Visto que as alterações do produto podem ocorrer de forma contínua, uma preocupação com o gerenciamento de *releases* é determinar quando publicá-lo. A gravidade dos problemas abordados pela entrega e as medições das densidades de falhas de versões anteriores afetam essa decisão. A tarefa de empacotamento deve identificar quais itens do produto devem ser entregues e, em seguida, selecionar as configurações corretas desses itens, dada a aplicação pretendida do produto.

As informações que documentam o conteúdo físico de um *release* são conhecidas como um documento de descrição de versão. As notas de lançamento descrevem tipicamente novas capacidades, problemas conhecidos e requisitos de plataforma necessários para a operação adequada do produto. O pacote a ser liberado também contém instruções de instalação ou atualização. Em alguns casos, o gerenciamento de *releases* pode ser necessário para rastrear a distribuição do produto a vários clientes ou sistemas de destino, por exemplo, no caso em que o fornecedor foi obrigado a notificar um cliente sobre problemas identificados.

Por fim, um mecanismo para garantir a integridade do item liberado pode ser implementado, por exemplo, liberando uma assinatura digital com o pacote.

7.8 Ferramentas de gerenciamento de configuração

As ferramentas do gerenciamento de configuração de software podem ser divididas em três tipos em relação ao escopo em que fornecem suporte:

- 1. ferramentas de suporte individual** – são ferramentas normalmente apropriadas para pequenas organizações ou grupos de desenvolvimento e incluem:
 - × controle de versão – acompanhar, documentar e armazenar itens de configuração individuais, como código-fonte e documentação;

- × controle de manipulação – em sua forma mais simples, comparam e vinculam uma versão executável do software; ferramentas de construção mais avançadas extraem a versão mais recente do software de controle de versão, executam verificações de qualidade, executam testes de regressão e produzem vários relatórios, entre outras tarefas;
 - × controle de mudanças – apoiam principalmente o controle de solicitações de mudança e notificação de eventos, por exemplo, mudanças de status de solicitação, metas alcançadas, entre outros.
2. **ferramentas de suporte ao projeto** – são ferramentas que fornecem principalmente suporte ao gerenciamento do espaço de trabalho para equipes de desenvolvimento e normalmente são capazes de suportar ambientes de desenvolvimento distribuídos; tais ferramentas são apropriadas para organizações de médio a grande porte, mas sem requisitos de certificação.
3. **ferramentas de suporte ao processo da empresa** – são ferramentas que normalmente podem automatizar partes de um processo de toda a empresa, fornecendo suporte para gerências de fluxo de trabalho, funções e responsabilidades, capazes de lidar com muitos itens, dados e ciclos de vida; essas ferramentas aumentam o apoio ao projeto, dando suporte a um processo de desenvolvimento mais formal, incluindo os requisitos de certificação.

7.8.1 Exemplos de ferramentas

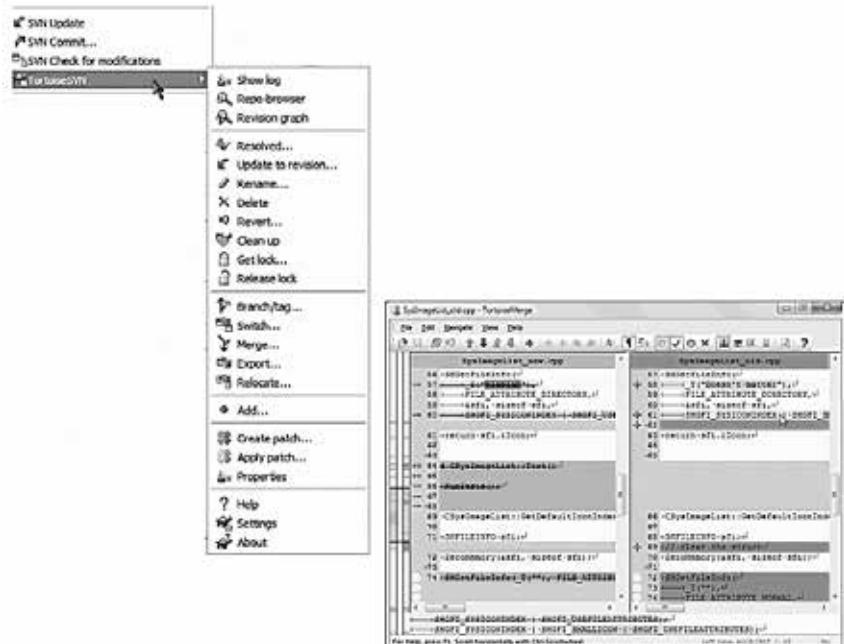
Subversion – SVN <<http://tortoisessvn.net>> é um sistema de controle de versão muito utilizado em projetos de software corporativos e grandes projetos Opensource, como SourceForge, Apache, Ruby e diversos outros, sendo amplamente utilizado em IDEs como Eclipse e Netbeans.

No Subversion, os arquivos não têm versões independentes, todos são parte de uma mesma revisão e a modificação de um único arquivo altera a revisão de todos. O Subversion utiliza o conceito de *Trunk*, *Branches* e *Tags*.

- × **Trunk:** uma pasta que contém os projetos que estão em desenvolvimento com as atualizações efetuadas no dia a dia;

- ✖ **Branches:** uma pasta que contém as “linhas de desenvolvimento” de um projeto, entre as quais existem diferenças: para cada *Branch* tem-se diferentes versões de um projeto. Quando a versão está pronta, migra-se a pasta *Trunk* para a pasta *Branch* e é dado um nome para ela. O *Branch* deve ser congelado e não sofrer mais alterações, apenas correções, se necessário. Isso é importante inclusive se for necessário voltar uma versão atrás no caso de uma versão em desenvolvimento estar enfrentando algum problema que levará certo tempo para ser arrumado.
- ✖ **Tags:** considerada apenas uma variação de um *Branch*, na prática é exatamente como um, apenas uma cópia da ramificação atual da árvore. A *Tag* é como uma versão liberada para o cliente após um *Branch* estar completo, é a pasta que deve ser empacotada e enviada para o cliente.

Figura 7.3 – Exemplo de utilização do SVN

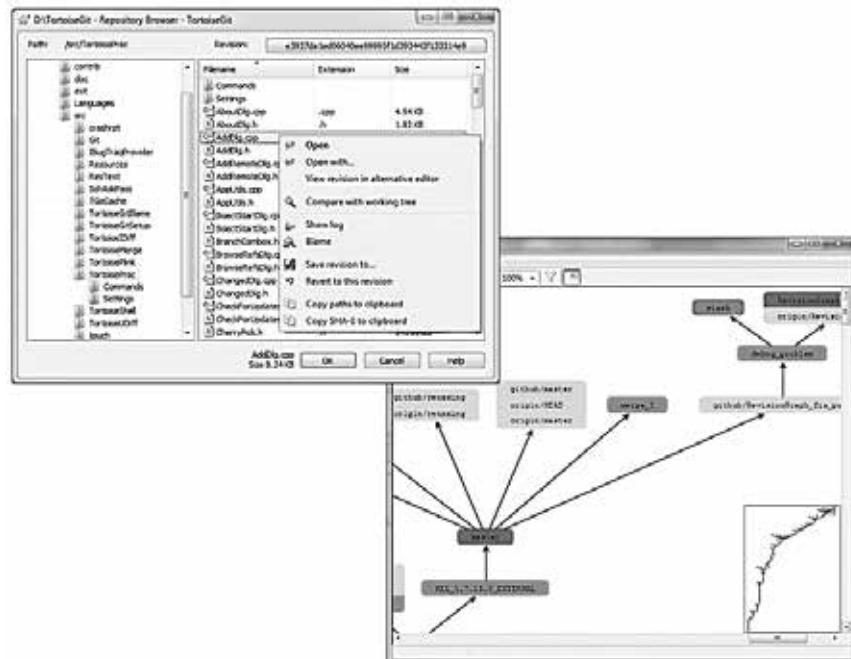


Fonte: Elaborada pelo autor.

O GIT <(site oficial <https://tortoisegit.org/>)> foi inicialmente desenvolvido pelo criador do Kernel do Linux Linus Tolvards. A diferença do GIT em relação ao SVN é que este não se baseia em uma base centralizada de código.

No GIT, diferentes pontas detêm partes diferentes do código, enquanto o SVN utiliza um controle centralizado, ou seja, apenas uma cópia original do software é utilizada. O GIT é rápido e eficiente utilizado, pelo Kernel do Linux, Fedora e diversos outros. Assim, cada desenvolvedor com o GIT em seu ambiente de desenvolvimento possui um repositório com todos os históricos e a habilidade total de controle das revisões, não dependente de acesso a uma rede ou a um servidor central.

Figura 7.4 -Exemplo de utilização do GIT



7.9 O gerenciamento de configuração nos modelos de qualidade de software

Todas as normas e os modelos de qualidade para software têm o objetivo de buscar organização e melhoria contínua no processo de desenvolvimento de software. Dessa maneira, o gerenciamento de configuração de software está totalmente ligado a essas normas e a esses modelos. Na sequência, verificaremos a visão do gerenciamento de configuração das principais normas e dos modelos de qualidade de software.

7.9.1 Gerenciamento de configuração de software no CMMI

O CMMI (Capability Maturity Model Integrated ou Modelo de Maturidade em Capacitação, em português) foi desenvolvido pelo Software Engineering Institute (SEI), ligado à Carnegie Mellon University (CMU), em Pittsburgh, nos Estados Unidos. Segundo Pessoa (2001), o desenvolvimento desse modelo foi financiado pelo Departamento de Defesa Americano, com o objetivo de se estabelecer um padrão de qualidade para software desenvolvido para as Forças Armadas.

O CMMI foi concebido para o desenvolvimento de grandes projetos militares e, para sua aplicação em projetos menores e em outras áreas, é necessário um trabalho cuidadoso de interpretação e adequação à realidade da organização. No modelo CMMI, foram estabelecidos níveis referentes à maturidade de organização para desenvolver softwares: inicial (1), gerenciado (2), definido (3), gerenciado quantitativamente (4) e otimizado (5). Cada nível de maturidade está dividido em áreas-chave de processo, chamadas de *Process Area* (PA), que estabelecem grandes temas a serem abordados, somando 18 áreas-chave.

No nível 2 do CMMI, os métodos de gerenciamento de software são documentados e acompanhados. Políticas organizacionais orientam os projetos estabelecendo processos de gerenciamento e práticas bem-sucedidas

podem ser repetidas em novos projetos. No nível 2 existe um sistema de gerenciamento implantado que garante o cumprimento de custos e de prazos em projetos similares já desenvolvidos.

Entre as PAs do nível 2 do CMMI está presente o gerenciamento da configuração de software, cujo propósito é estabelecer e manter a integridade de produtos usando a identificação de configuração, controle de configuração, condição de configuração e auditorias de configuração (SEI). A área do processo de gerenciamento de configuração envolve:

- × identificar a configuração de produtos selecionados que compõem as *baselines* em pontos dados no tempo;
- × controlar mudanças para artigos de configuração;
- × construir ou prover especificações dos produtos de software;
- × manter a integridade das *baselines*;
- × prover condição e dados precisos de configuração atuais para desenvolvedores, usuários finais e clientes.

7.9.2 Gestão de configuração na norma ISO 12207

A norma internacional *NBR ISO/IEC 12207 – Tecnologia da Informação – Processos de Ciclo de Vida de Software* é usada como referência em muitos países para alcançar o diferencial competitivo. Tem por objetivo auxiliar os envolvidos na produção de software a definir seus papéis por meio de processos bem definidos e assim proporcionar às organizações que a utilizam um melhor entendimento das atividades a serem executadas nas operações que envolvem, de alguma forma, o software.

A arquitetura descrita na norma utiliza uma terminologia bem definida e é composta de processos, atividades e tarefas para aquisição, fornecimento, desenvolvimento, operação e manutenção do software. Está estruturada em três processos de ciclo de vida (ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS, 1998):

1. processos fundamentais;
2. processos de apoio;
3. processos organizacionais.

A gestão de configuração de software está presente nos processos de apoio de ciclo de vida, que auxiliam e contribuem para o sucesso e a qualidade do projeto de software. Um processo de apoio é empregado e executado quando necessário para documentação, gerência de configuração, garantia da qualidade, processo de verificação, processo de validação, revisão conjunta, auditoria e resolução de problemas.

A norma ISO 12207 descreve que o processo de gerência de configuração consiste em:

- × implementação do processo;
- × identificação da configuração;
- × controle da configuração;
- × relato da situação da configuração;
- × avaliação da configuração;
- × gerência de liberação e distribuição.

7.9.3 Gestão de configuração de software na ISO 9000-3

A ISO 9000 é composta de uma série de normas e reconhece que existem quatro categorias genéricas de produtos e publicou diretrizes para implementação de sistemas da qualidade em cada uma dessas categorias: Hardware: ISO 9004-1; Serviços: ISO 9004-2; Materiais Processados: ISO 9004-3; e Software: ISO 9000-3.

De acordo com Xavier (2001), devido às dificuldades específicas de interpretação sobre como implantar os requisitos da ISO 9001 ou 9002 em software, é fundamental o uso da ISO 9000-3 para auxiliar a implantação do sistema de gestão da qualidade. Essa dificuldade está relacionada com a terminologia usada na ISO 9001, muito voltada para hardware; com a ISO 9000-3, essa barreira é eliminada.

As normas da série ISO 9000 foram expandidas com a edição das normas internacionais de série 10.000, que as complementam e fornecem diretrizes para planos de qualidade, auditorias internas, qualificação de auditores, comprovação metrológica e manuais de qualidade. Nessa expansão, foi criada

uma norma específica para o Gerenciamento da Configuração, a ISO 10.007

– Gestão da qualidade – Diretrizes para gestão da configuração. A norma ISO 10.007 fornece diretrizes para gestão da configuração, uma disciplina aplicada sobre o ciclo de vida de um produto a fim de fornecer visibilidade e controle de suas funcionalidades e características físicas. As atividades descritas são uma forma de satisfazer certos requisitos encontrados em outras normas da série ISO 9000.

Na norma ISO 9000-3, o gerenciamento de configuração está inscrito no requisito *Identificação e Rastreabilidade* e consiste em:

- × quando apropriado, a organização deve identificar o produto por meios adequados ao longo da realização do produto; a organização deve identificar a situação do produto no que se refere aos requisitos de monitoramento e de medição;
- × Quando a rastreabilidade é um requisito, a organização deve controlar e registrar a identificação única do produto.

7.9.4 Gestão de configuração de software no projeto Spice

A ISO/IEC 15504, Spice (Software Process Improvement and Capability Etermination), presta-se à realização de avaliações de processos de software com dois objetivos: melhoria dos processos e determinação da capacidade de processos de uma organização. De acordo com Salviano (2001), se o objetivo for a melhoria dos processos, a organização pode realizar a avaliação gerando um perfil dos processos que serão usados para essa elaboração. A organização deve definir os objetivos e o contexto, bem como escolher o modelo e o método para a avaliação e definir os objetivos de melhoria.

No segundo caso, a empresa tem o objetivo de avaliar um fornecedor em potencial, obtendo seu perfil de capacidade. Para isso, deve definir os objetivos e o contexto da avaliação, os modelos e os métodos de avaliação e os requisitos esperados. O perfil de capacidade permite ao contratante estimar o risco associado à contratação daquele fornecedor em potencial para auxiliar na tomada de decisão de contratá-lo ou não (SALVIANO, 2001).

A norma ISO 15504 foi baseada nos principais modelos existentes como CMMI, Trillium e Bootstrap, tendo entre objetivos básicos realizar uma harmonização desses diversos modelos existentes e prover um *framework* para esses e outros modelos que possam ser criados.

O modelo de referência descreve um conjunto de processos e boas práticas da Engenharia de Software considerados universais e fundamentais. Quarenta processos e componentes de processos são descritos e organizados em cinco categorias de processo: cliente-fornecedor, engenharia, suporte, gerência e organização. Esses processos são um conjunto dos processos definidos na ISO/IEC 12207 e a gerência de configuração está também presente no Spice, na categoria Suporte denominado processo de gerência de configuração.

Síntese

Vimos neste capítulo que, devido às constantes mudança que permeiam o ciclo de vida do desenvolvimento de software, são necessárias formas de organização e controle de todos os elementos envolvidos nessas mudanças.

Assim, o gerenciamento de configuração de software surge como uma solução que beneficia a gestão do projeto, as atividades de desenvolvimento e de manutenção, as atividades de garantia de qualidade e até mesmo os clientes e os usuários do produto final. Uma vez que o gerenciamento de configuração direciona como estruturar, gerenciar e controlar o software por todo seu ciclo de vida, proporcionando acompanhamento de todo o processo.

Verificamos que as atividades de configuração de software basicamente são planejamento do gerenciamento de configuração de software, identificação de configuração de software, controle de configuração de software, registro de status de configuração de software, auditoria de configuração de software e gerenciamento e entrega do software.

Por fim, ainda apresentamos a visão do gerenciamento de configuração mediante as principais normas e modelos de qualidade de software: CMMI, ISO 12207 e ISO 9000-3.

Atividades

1. De que se trata o gerenciamento de configuração de software, na Engenharia de Software?
2. O que é um item de configuração de software? Cite alguns exemplos.
3. O que significa o termo *baseline* para a configuração de software?
4. Quais são tipos de auditorias formais normalmente encontradas na indústria de software? E qual é o objetivo de cada uma?

8

Gerenciamento de Engenharia de Software

É OBJETIVO DA engenharia de software auxiliar as organizações desenvolvedoras de software na construção deste de maneira profissional e planejada, para evitar o risco de sofrer com o alto custo e o tempo que o projeto exige para ser elaborado. Além disso, a engenharia de software busca evitar que o mercado perca o interesse pelo produto e também verifica se a viabilidade de produção compensará o custo do produto.

UMA DAS ALTERNATIVAS seria um gerenciamento de projeto adequado, para que a organização seja capaz de atender às exigências dos clientes com maior qualidade e com um preço competitivo, e, assim, a organização se destacaria perante o mercado cada vez mais exigente.

ESTE CAPÍTULO TEM como objetivo introduzir os conceitos sobre o gerenciamento de software em seus aspectos fundamentais.

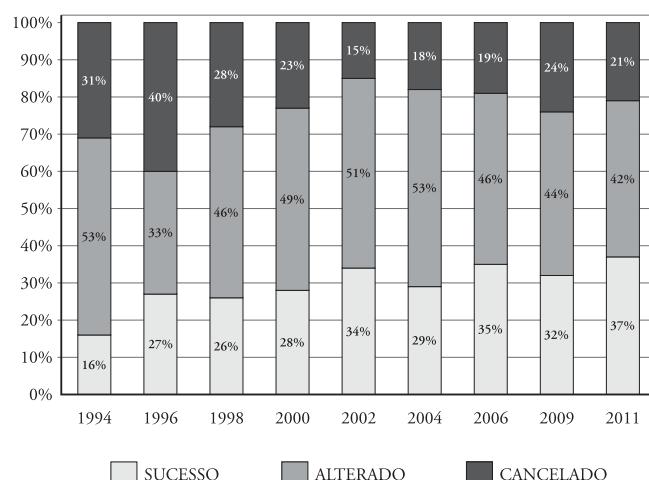
8.1 Fundamentação do gerenciamento de engenharia de software

O gerenciamento de projetos é a aplicação de conhecimentos, habilidades e técnicas para a execução de projetos de modo eficaz, permitindo conciliar o resultado dos projetos com os objetivos do negócio (PMI, 2013).

Por esse motivo, muitas empresas, de diversos segmentos, vêm reconhecendo a importância dessa disciplina para o sucesso dos seus negócios, ou seja, para a competência estratégica nas organizações. A prova disso são os estudos realizados pelo Standash Group para avaliar o desempenho dos projetos de Tecnologia da Informação (TI), cujos resultados têm indicado índices de insucesso. Os tipos de situações apontadas pela pesquisa estão destacados a seguir.

- ✗ Projetos com sucesso: projetos entregues no prazo, no orçamento, com recursos necessários e todas as funções originalmente previstas.
- ✗ Projetos alterados: projetos atrasados, acima do orçamento e/ou com menos características e funções requeridas.
- ✗ Projetos cancelados: projetos não concluídos ou abandonados.

Figura 8.1 – Desempenho de projetos de TI



Fonte: Adaptado de Davidson (2012 apud CHAOUS, 2012).

Um projeto de sucesso na visão do PMI (*Project Management Institute* – que utiliza as melhores práticas de gerenciamento de projetos) pode ser medido em termos da sua conclusão dentro das restrições de escopo, tempo, custo, qualidade, recursos e riscos. O sucesso do projeto deve referir-se às últimas linhas de base aprovadas pelas partes interessadas.

Conforme a figura 8.1, nota-se que as taxas dos projetos com sucesso vêm crescendo desde 1994; entretanto, os valores ainda são baixos, ficando sempre inferiores a 40%. Com relação aos projetos alterados, ou seja, os que foram concluídos, mas com alterações, o percentual está na média dos 50%. Por fim, projetos cancelados apresentam um percentual variável, com tendência a se concentrar em torno de 20%.

Esse cenário mostra a importância de se utilizar um conjunto de processos e técnicas do gerenciamento de projetos, visando atingir os objetivos propostos inicialmente, dentro do prazo estipulado e com os custos esperados, evitando desperdício de tempo e dinheiro.

Na área de TI, a aplicação dessa prática é tão importante como em qualquer outra área de negócio, seja de pequeno, médio ou grande porte, com forte referência no PMI (PMI, 2013). Este vem se tornando um grande diferencial competitivo para as empresas que almejam obter sucesso nos serviços prestados em um mercado cada vez mais exigente.

Os projetos de TI precisam ser gerenciados, pois a engenharia de software profissional está sujeita às restrições de orçamento e cronograma. O objetivo é diminuir as chances de fracasso do projeto, pois o mau gerenciamento de projeto costuma resultar em falha de projeto ou até mesmo em um impacto mais grave e irremediável para o negócio.

A gerência da engenharia de software pode ser definida como a aplicação das atividades de gerenciamento, planejamento, coordenação, mensuração, monitoração, controle e informação, de modo a garantir que o desenvolvimento e a manutenção do software sejam sistemáticos, disciplinados e quantificados (IEEE, 2004).

No entanto, existem aspectos específicos aos produtos de software e aos processos do ciclo de vida do software que complicam o gerenciamento efetivo, alguns dos quais destacados a seguir (SOMMERVILLE, 2011).

- ✗ O produto é intangível: em um projeto de engenharia civil, por exemplo, o progresso da obra é facilmente verificável no cronograma – partes da estrutura inacabada são fáceis de visualizar. Já a visualização do progresso dos projetos de TI é mais difícil – simplesmente olhando para o artefato que está sendo construído não se tem a real dimensão do trabalho. Dependem de outros artefatos para revisar o acompanhamento do projeto. Assim, devido a essa característica, há uma falta de reconhecimento da complexidade por parte do cliente inerente à engenharia de software, particularmente em relação ao impacto das mudanças dos requisitos.
- ✗ Os grandes projetos de software são, muitas vezes, projetos únicos: geralmente, os projetos de TI são diferentes dos projetos anteriores em algum aspecto, o que dificulta a análise de riscos até para os gerentes de projeto mais experientes. Além disso, as rápidas mudanças tecnológicas podem tornar obsoleta a experiência do gerente de projeto. Assim, as lições aprendidas de um projeto podem não servir para os futuros projetos.
- ✗ Os processos de software são variáveis e de organização específica: os processos de desenvolvimento para algumas áreas, como a engenharia civil, que constrói pontes e edifícios, podem ser bem compreendidos. No entanto, para o contexto de projetos de software, um processo de desenvolvimento pode variar de uma empresa para outra.

Embora existam características específicas para os projetos de software que dificultam seu gerenciamento, os gerentes de projetos, guiados pela experiência e pela prática de projeto, focam em atividades como planejamento, gerenciamento de riscos, gerenciamento de pessoas, atividades de monitoramento, como a geração de relatórios.

De acordo com o SWEBOK, as atividades de gerenciamento ocorrem em três níveis: gerenciamento organizacional e de infraestrutura, gerência de projetos e planejamento e controle do programa de mensurações (IEEE, 2004).

Aspectos de gerência organizacional são importantes devido à definição de políticas organizacionais que impactam na forma como os projetos de software são desenvolvidos, ou seja, definição de padrões de desenvolvimento, e fornecem a estrutura na qual a engenharia de software é realizada. Alguns

exemplos desse nível são: estabelecer processos específicos em escala organizacional ou procedimentos para tais tarefas de engenharia de software, como projeto, implementação, estimativas e monitoramento (IEEE, 2004).

Os dois últimos aspectos – gerência de projetos e planejamento – também são aspectos importantes para o sucesso dos projetos e encontram-se no PMBOK® (Guia do Conhecimento em Gerenciamento de Projetos), diretrizes reconhecidas mundialmente com relação a boas práticas e metodologias de gerenciamento de projeto.

O Guia PMBOK® é um guia globalmente reconhecido para a profissão de gerenciamento de projetos (PMI, 2013). É um documento formal que descreve normas, métodos, processos e práticas estabelecidos. Assim como em outras profissões, o conhecimento contido nesse padrão evoluiu a partir das boas práticas reconhecidas por profissionais de gerenciamento de projetos que contribuíram para o seu desenvolvimento.

8.1.1 O que é um projeto?

Este capítulo tem abordado a importância da gerência de projetos. Então, faz-se necessário o entendimento sobre o que é um projeto. A definição dada pelo o PMI (2013, p. 3) é que projeto é “um esforço temporário empreendido para criar um produto, serviço ou resultado exclusivo”.

Segundo Vargas (2002), projeto é “um empreendimento não repetitivo, caracterizado por uma sequência clara e lógica de eventos, com início, meio e fim, que se destina a atingir um objetivo claro e definido, sendo conduzido por pessoas dentro de parâmetros definidos de tempo, custo, recursos envolvidos e qualidade”.

De acordo com as definições apresentadas, as características de projeto podem ser classificadas do modo descrito a seguir.

1. Básica

- ✗ Temporário: tem data de início de término definidos.
- ✗ Resultado exclusivo: significa que o produto, o serviço ou o resultado produzido é diferente, de alguma forma, de todos os outros produtos ou serviços semelhantes.

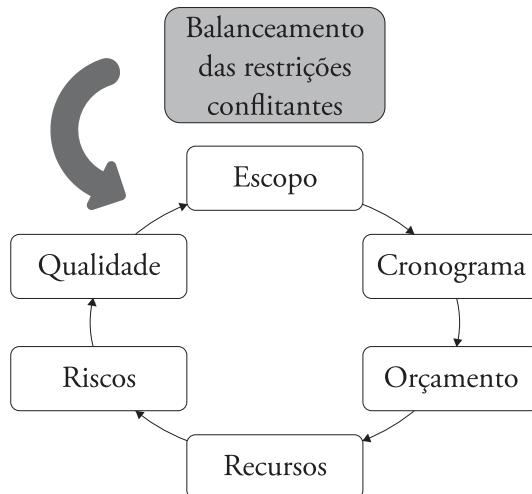
2. Adicional

- ✗ Tem objetivo específico que pode ser concluído.
- ✗ É restrito a recursos limitados.
- ✗ Envolve várias áreas do conhecimento.

8.1.2 O que é gerenciamento de projetos?

Conforme o PMI (2013), gerenciamento de projetos “é a aplicação de conhecimento, habilidades, ferramentas e técnicas às atividades do projeto a fim de atender aos seus requisitos”. O objetivo da gerência de projetos é diminuir os riscos de fracasso do projeto e auxiliar os gerentes de projeto no balanceamento (controle) das restrições conflitantes de projeto (figura 8.2).

Figura 8.2 – Função da gestão de projetos



Fonte: Adaptado de PMI (2013).

Mas por que essas áreas entram em conflito durante a execução dos projetos?

Em qualquer projeto, a alteração de requisitos pode afetar os demais aspectos, como custos ou prazo. Por exemplo, imagine a construção de uma

casa; no decorrer da execução, o cliente resolve que precisa de mais um quarto, o que significa gastar mais tempo e envolve mais custos. Para esse novo quarto, terá de ter mais material de construção e também mais mão de obra. Além dos custos adicionais, pode ser que o quarto não se adeque perfeitamente ao layout já definido, ou seja, os riscos de projeto aumentam e talvez seja necessário alterar a estrutura da casa e afetar partes já construídas, o que pode afetar também a qualidade. Então, qualquer alteração em alguns desses fatores vai impactar nos demais fatores.

Conforme apresentado no exemplo, se no escopo do projeto são adicionados requisitos, consequentemente, aumenta a quantidade de riscos e recursos, o que afeta o cronograma e o orçamento.

De modo a auxiliar os gerentes de projeto em sua tarefa de gerenciamento, o PMBOK sugere a aplicação de 47 processos. Esses processos estão divididos em 5 grupos: iniciação; planejamento; execução; monitoramento e controle, e encerramento.

São diretrizes relacionadas para qualquer tipo de projeto e são importantes de serem mencionadas. Mas nem todas as práticas se aplicam a todos os tipos de projeto. Cabe ao gerente de projetos definir os tópicos importantes para realizar a gestão adequada.

Vale mencionar que não é intenção deste capítulo apresentar extensivamente o conteúdo desses processos, mas apresentar as principais atividades relacionadas a projetos de software na visão do SWEBOK.

8.1.3 Atividades de gerência de engenharia de software

Segundo Sommerville (2011), é impossível a padronização de atividades gerenciais para projetos de software em virtude das próprias características do software. Mas existem algumas atividades que podem auxiliar o gerente de projetos durante o desenvolvimento de software.

O SWEBOK recomenda sete tópicos (IEEE, 2004):

- * **iniciação e definição de escopo**, relacionada às atividades para iniciar um projeto de engenharia de software;

- × **planejamento do projeto de software**, que orienta as atividades empreendidas para preparar para o sucesso da engenharia de software a partir de uma perspectiva de gerenciamento;
- × **formalização do projeto de software**, que aborda as atividades de gerência de engenharia de software geralmente aceitas, que ocorrem durante a engenharia de software;
- × **análise e avaliação**, que trata da garantia de que o software seja satisfatório;
- × **fechamento (encerramento)**, que trata das atividades de pós-realização de um projeto de engenharia de software;
- × **mensuração da engenharia de software**, que aborda o desenvolvimento e a implementação efetiva de programas de mensuração nas organizações de engenharia de software;
- × **ferramentas de gerenciamento de engenharia de software**, que descreve a seleção e o uso de ferramentas para gerenciar um projeto de engenharia de software.

8.2 Iniciação e definição de escopo

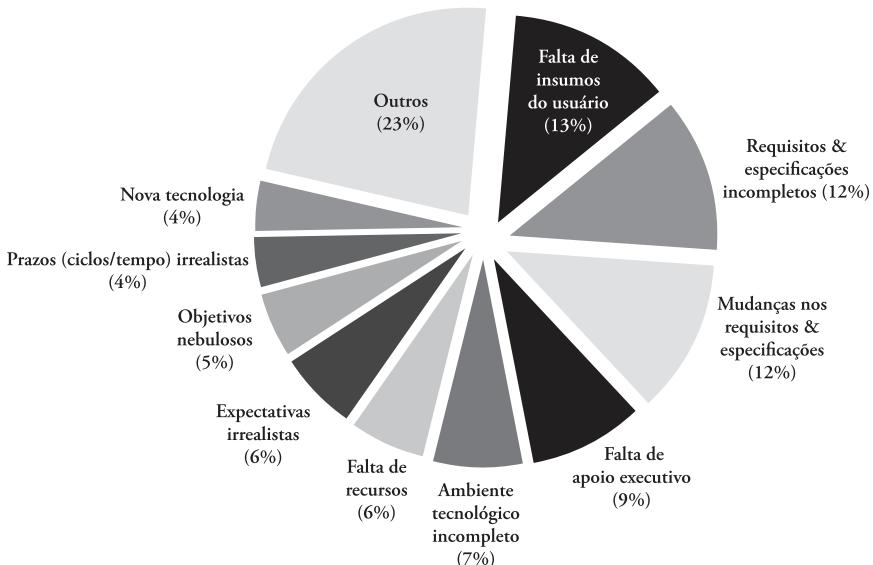
Esta seção trata das atividades relacionadas à definição do escopo do projeto de software. Portanto, apresenta atividades para determinação dos requisitos de software usando vários métodos de elicitação e a avaliação da viabilidade do projeto, bem como as atividades de revisão dos requisitos.

8.2.1 Determinação e negociação de requisitos

Os requisitos de um sistema são as descrições do que ele deve fazer, os serviços que ele oferece e as restrições de seu funcionamento. Os requisitos são importantes para os projetos de software, pois são a base para análise de contratos, elaboração/análise de propostas, planejamento, estimativas, gestão e controle, modelagem do sistema e desenvolvimento.

Segundo pesquisas realizadas pelo Standish Group, os requisitos de software têm um papel central no processo de software, sendo considerados fatores determinantes para o sucesso ou fracasso de um projeto de software. Falhas em sua especificação são apontadas como uma das principais fontes da causa no fracasso de um projeto (figura 8.3).

Figura 8.3 – Causas da falha de projetos

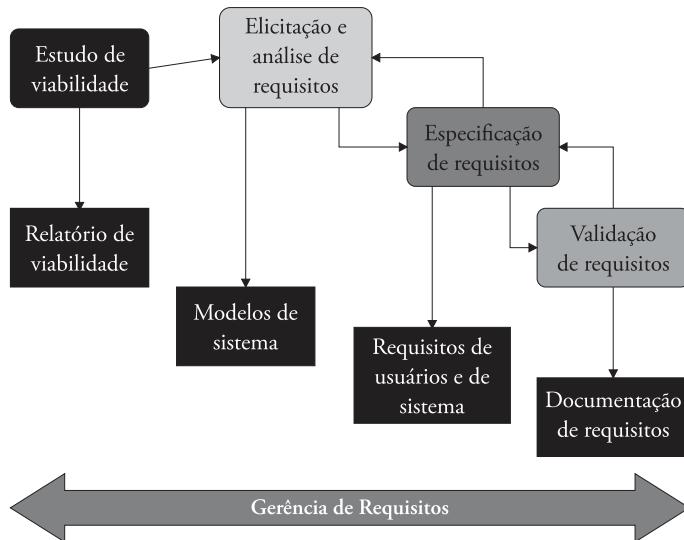


Fonte: Chaos Report (2015).

Conforme apresentado no gráfico da figura 8.3, um dos fatores que podem influenciar no entendimento dessas necessidades é identificação e envolvimento de todas as partes interessadas do projeto de software (cliente, patrocinador e outras). Percebe-se, portanto, que o entendimento das necessidades dos clientes é uma das tarefas mais complexas na execução dos projetos de software.

De maneira a evitar uma compreensão incorreta a respeito do levantamento das necessidades do projeto de software, é importante a execução de um processo de engenharia de requisitos. A seguir, a figura 8.4 ilustra o processo de engenharia de requisitos proposta por Sommerville (2011).

Figura 8.4 – Processo de engenharia de requisitos



Fonte: Adaptado de Sommerville (2011).

O **estudo de viabilidade** analisa se o projeto de software é viável ou não de ser realizado. É um estudo que apresenta uma análise mais detalhada e verifica os seguintes pontos:

- ✗ se o sistema contribui para os objetivos da organização;
- ✗ o escopo do produto;
- ✗ se o sistema pode ser implementado usando tecnologia atual;
- ✗ estimativa de prazo, custos e recursos necessários;
- ✗ os riscos iniciais do projeto;
- ✗ se o sistema pode ser integrado a outros.

A etapa de elicitação e análise de requisitos é o processo de recolha das necessidades dos usuários. Nessa etapa, são feitas discussões com os usuários e são aplicadas as técnicas de elicitação que ajudam a entender o sistema a ser especificado. Algumas das técnicas citadas no PMBOK (PMI, 2013)

são: entrevistas; oficinas facilitadas (workshops); técnicas de criatividade em grupo (brainstorming, técnica do grupo nominal e outras); questionários; pesquisas em documentos; observações; protótipos, entre outras.

A etapa de **especificação de requisitos** tem como principal resultado um documento de requisitos que oficializa as informações levantadas na etapa de elicitação de requisitos. Segundo Sommerville (2011), podem ser incluídos dois tipos de requisitos: requisitos de usuários (declarações mais abstratas) e requisitos de sistema (declarações mais detalhadas).

Nessa fase, os requisitos são analisados e priorizados pelas partes interessadas do projeto. A análise é importante devido a inconsistências de requisitos oriundos da fase de elicitação (diferentes fontes de informação). Além disso, inevitavelmente, os diferentes stakeholders têm opiniões diferentes (por vezes, conflitantes) sobre a importância dos requisitos. Por isso, é importante que o gerente de projetos negocie com todas as partes interessadas a priorização dos requisitos e juntos cheguem a um consenso a respeito do escopo do projeto.

A **validação dos requisitos** verifica os requisitos quanto a realismo, consistência e completude. É a verificação da lista de requisitos em que os erros identificados são corrigidos. Nessa fase, é importante identificar os possíveis problemas nos requisitos antes da fase de implementação do sistema a fim de descobrir problemas, omissões e ambiguidades nos requisitos.

Durante a execução do projeto de software podem ocorrem mudanças nos requisitos, além do surgimento de novos. Por isso, o gerenciamento de requisitos é necessário para minimizar as consequências das mudanças. O **gerenciamento de requisitos** envolve as atividades de controle do escopo do projeto e gerência das mudanças nos requisitos de um sistema. Cada mudança deve ser: avaliada; estimada; aprovada; controlada e documentada.

Além disso, é fundamental um acordo entre as partes interessadas no início do projeto, com relação às solicitações de mudanças. As exigências e o acordo devem ser revisados (por exemplo, por meio de mudanças na gestão dos procedimentos). No entanto, para facilitar a análise de impacto da mudança no projeto, é importante realizar a rastreabilidade nos requisitos (IEEE, 2004).

8.3 Planejamento do projeto

Uma vez definido o escopo do projeto é importante fazer o planejamento dele. Um plano de projeto deve conter todas as informações necessárias para sua realização, como o detalhamento das atividades dos entregáveis, atribuição de atividades aos membros da equipe, além da identificação dos riscos de projeto e definição das estratégias adequadas para o tratamento, bem como a definição de plano de comunicação e qualidade do projeto.

Segundo o SWEBOK (IEEE, 2004), o primeiro passo no planejamento de projetos de software deve ser a seleção de um modelo de ciclo de vida de desenvolvimento de software apropriado, a adaptação dele com base no escopo do projeto, nos requisitos de software, e uma avaliação de risco.

8.3.1 Planejamento do processo e das entregas

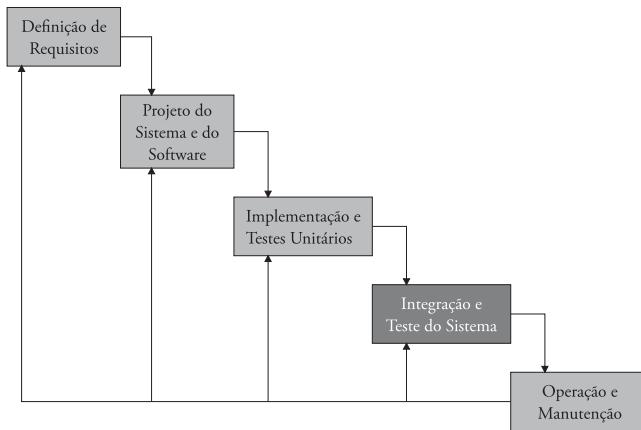
Um processo de software fornece a metodologia por meio da qual um plano de projeto abrangente para o desenvolvimento de software pode ser estabelecido (PRESSMAN, 2011).

Os ciclos de vida do projeto podem variar ao longo de uma sequência contínua, desde abordagens previsíveis ou direcionadas por um plano em uma extremidade, até abordagens adaptativas ou acionadas por mudanças na outra (PMI, 2013).

Em um ciclo de vida previsível (figura 8.5), as entregas são definidas no início do projeto e quaisquer mudanças no escopo são cuidadosamente gerenciadas.

Os preditivos são caracterizados pelo desenvolvimento de requisitos de software detalhados, planejamento de projeto detalhado e planejamento mínimo para iteração entre fases de desenvolvimento. Alguns já são bem conhecidos, como cascata, RUP e outros, que são denominados abordagens dirigidas a planos. Geralmente são escolhidos pelo gerente de projetos quando o produto é bem definido e tem que realizar uma única entrega para os usuários.

Figura 8.5 – Exemplo de ciclo de vida previsível



Fonte: Adaptado (PMI, 2013).

Em um ciclo de vida adaptativo, o produto é desenvolvido por meio de múltiplas iterações, e um escopo detalhado é definido para cada iteração somente no seu início. Os métodos adaptativos geralmente são preferidos quando se lida com um ambiente em rápida mutação, quando requisitos e escopo são difíceis de definir antecipadamente e quando é possível definir pequenas melhorias incrementais que entregará valor às partes interessadas (PMI, 2013). Alguns processos bem conhecidos são SCRUM e Extreme Programming (XP).

A partir da definição do ciclo de vida do projeto, são definidas as entregas deste. Os produtos de cada tarefa (por exemplo, projeto de arquitetura, relatório de inspeção e outros) são especificados e caracterizados (IEEE, 2004).

8.3.2 Programação e custos do projeto

Após a definição das atividades das entregas do projeto, faz-se necessário a definição de cronograma e orçamento para fins de contrato e gerenciamento do projeto.

A programação de um projeto é um processo de decidir como será organizado o trabalho em tarefas separadas e quando e como essas tarefas serão executadas (SOMERVILLE, 2011). Para isso, é feita uma análise de sequenciamento das tarefas (definição de dependências entre as atividades). Também são estimados os recursos necessários para conclusão de cada atividade, o que envolve uma análise das necessidades e disponibilidades de recursos, que podem ser: equipamentos, instalações e profissionais associadas às tarefas agendadas, além da atribuição de papéis e responsabilidades e habilidades técnicas para a conclusão. Com base na definição do calendário de recursos, informações de bases históricas e/ou opinião especializada são estimados os prazos das atividades. O resultado desse processo é o cronograma do projeto, que apresenta a conexão das atividades com datas, durações, marcos e recursos planejados.

Outro ponto importante no plano de projeto é a estimativa dos recursos financeiros necessários para executar as atividades do projeto. Geralmente, é baseada nas atividades do cronograma e no calendário de recursos humanos e pode ser utilizado o valor da hora da produtividade da empresa (por exemplo, imaginemos que a empresa produza um faturamento de R\$ 50.000,00 por mês com 10 funcionários trabalhando 160 horas/mês cada; dessa forma, podemos calcular que $10 \times 160\text{hs} = 1.600\text{ hs/mês}$ “disponível”; assim $50.000 / 1.600 = 31,25\text{ R$/h}$) ou os valores reais dos recursos (por exemplo, vamos supor que o recurso do projeto receba o salário + encargos por mês no valor de R\$ 2.000,00 e trabalhe 160 horas/mês; assim, calculamos $2.000 / 160 = \text{R\$ } 12,50\text{ R$/h}$).

É importante que, para o cálculo total do orçamento, o gerente de projetos considere os custos diretos (por exemplo, mão de obra direta e/ou indireta, materiais, equipamentos) e os custos indiretos do projeto (aluguel das instalações da empresa, licenças e permissões, custos administrativos, etc.).

8.3.3 Gestão de pessoas

A gerência dos recursos humanos descreve os processos que organizam e gerenciam a equipe do projeto (PMI, 2013).

Sendo uma das principais fontes da produtividade no desenvolvimento de sistemas, a gestão e o planejamento das pessoas envolvidas no projeto são

atividades fundamentais para que não haja problemas quanto ao prazo. Por isso, a equipe de desenvolvimento é considerada um importante elemento para o sucesso dos projetos.

Atualmente a importância é cada vez mais percebida pelas organizações de TI e, por isso, há ênfase na gestão de pessoas e na busca por profissionais competentes. Para Silva (2011), no que se refere à seleção da equipe de projeto, a qualidade é sempre um fator crítico de sucesso de um projeto. Sendo a equipe um conjunto de indivíduos singulares, o desafio é fazer a diversidade produzir resultados por meio da combinação correta dos estilos de cada um. Avaliar bem as características individuais de cada integrante poderá gerar resultados poderosos quanto ao desempenho do projeto e atingir níveis de excelência salutares.

Confirmando essa visão, o Chaous Report indicou que um dos fatores críticos para o sucesso dos projetos de TI seria a formação de uma equipe competente para o desenvolvimento das atividades (STANDISH GROUP, 2015). Nesse sentido, o trabalho em equipe é um fator que influencia o êxito do projeto, e, portanto, o gerente de projetos deve criar um ambiente propício que facilite o trabalho em equipe, motivar a equipe continuamente fornecendo desafios e oportunidades, oferecendo feedback e apoio conforme necessário e, ao mesmo tempo, reconhecendo o bom desempenho.

Entretanto, Sommerville (2011) diz que geralmente bons gerentes de software têm boas habilidades técnicas, mas não necessariamente são bons com gestão de pessoas. Pressman (2011) apresenta características de um bom líder de equipe definidas no livro de Jerry Weinberg (*On Becoming a Technical Leader*):

- ✖ motivação – habilidade para encorajar o pessoal técnico a produzir o melhor da sua habilidade;
- ✖ organização – habilidade para moldar os processos já existentes (ou inventar novos) que irão capacitar o conceito inicial para ser traduzido num produto final;
- ✖ ideias ou inovação – habilidade de encorajar pessoas para criar e serem criativas mesmo quando estiverem trabalhando de acordo com padrões estabelecidos para um produto ou aplicativo de software específico.

Segundo o autor citado, líderes de projeto bem-sucedido utilizam uma abordagem de gerenciamento de solução de problemas, em que há o entendimento do problema a ser resolvido, gerenciando o fluxo de ideias e, ao mesmo tempo, deixam claro para equipe a importância da qualidade.

A equipe precisa resolver os problemas com os outros *stakeholders* do projeto e informá-los das mudanças do sistema e dos planos de entregas. Para isso, é importante que a comunicação seja eficaz e eficiente entre os membros da equipe e os demais envolvidos.

8.3.4 Gestão de riscos

O risco é um evento ou uma condição incerta que, se ocorrer, poderá impactar positivamente ou negativamente um projeto ou pelo menos um objetivo do projeto (PMI, 2013).

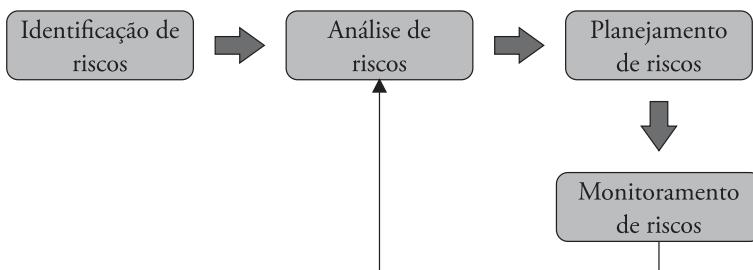
É importante que o gerente de projeto considere em seu planejamento os prováveis riscos do projeto e defina estratégias para lidar com esses imprevistos. Segundo Pressman (2011), os riscos podem ser classificados em: riscos de projeto, riscos técnicos e riscos de negócio.

- × **Riscos de projetos** são os que terão impacto no plano de projeto. Os potenciais riscos são: escopo (mudanças nos requisitos); cronograma e orçamento (estimativas erradas) e pessoal (equipes, organização, problemas técnicos e conflitos).
- × **Riscos técnicos** são os que ameaçam a entrega e a qualidade do software a ser desenvolvido. Os principais riscos são: análise, design, implementação e testes; ferramentas de hardware e software e indisponibilidade de infraestrutura.
- × **Riscos de negócios** são os que ameaçam a viabilidade do software a ser desenvolvido e, muitas vezes, ameaçam o produto ou o projeto. Os principais riscos são: risco de mercado (criar um produto que ninguém quer); risco estratégico (criar um produto que difere da estratégia geral da organização); risco de vendas (criar um produto que a equipe de vendas não sabe como vender); risco gerencial (perda de apoio gerencial da alta direção devido à mudança de foco da organização), entre outros.

Um plano de riscos considera os seguintes elementos: definição de metodologias para gerenciamento de riscos; definição de papéis e responsabilidades para monitoramento dos riscos do projeto; estimativa de orçamento para contingência de riscos; prazo (definição da frequência de acompanhamento das atividades de gerenciamento de riscos); categorização dos riscos de projeto (agrupamento das possíveis causas de riscos); definição da probabilidade e impacto dos riscos identificados (priorização dos riscos de projeto); definição de formatos de relatórios e acompanhamento ao longo do projeto (PMI, 2013).

O processo de gerenciamento de riscos é iterativo porque novos riscos podem ser conhecidos conforme o projeto se desenvolve durante todo o seu ciclo de vida. A seguir, a figura 8.6 ilustra um processo de gestão de riscos.

Figura 8.6 – Processo de gerenciamento de riscos



Fonte: Adaptado de Somerville (2011).

A atividade de identificação de riscos determina o que pode afetar o projeto e documenta suas características. Alguns riscos já podem ser identificados logo no início do projeto e outros são identificados ao longo do projeto. Podem participar dessa atividade o gerente de projetos, a equipe técnica e, como alternativa, um especialista no assunto do projeto. Em seguida, deve-se realizar uma análise de impacto dos riscos para gerar uma lista de priorização de riscos, a partir de uma combinação de sua probabilidade (alta, média e baixa) de ocorrência e impacto (catastróficos, toleráveis e insignificantes) em algum resultado do projeto. Após a priorização dos riscos, é feito o planejamento de riscos, que é o processo de definição de estratégias para tratar os riscos, bem como a definição de uma pessoa responsável para o monitoramento dos riscos.

Algumas estratégias de riscos são propostas como: evitar o risco, mitigar o risco e conter o risco (PRESSMAN, 2011).

Evitar o risco é uma ação proativa da equipe e é sempre a melhor estratégia. No entanto, é necessário fazer um plano de mitigação, que são ações a serem tomadas para prevenir ou eliminar os riscos antes de ocorrem, também denominadas ações preventivas. Quando aplicar? Quando os custos da consequência são maiores que os custos da prevenção. Exemplo de situação: evitar erros na duração das atividades:

- × consultar profissionais experientes em projetos semelhantes;
- × montar cenários pessimistas e otimistas.

O plano de contingência deve contemplar ações a serem tomadas após a ocorrência do risco para conter os efeitos deste no projeto, denominadas ações corretivas. Exemplo: contratação imediata de profissional qualificado.

8.3.5 Gestão da qualidade

A qualidade é considerada um requisito vital dos produtos de software, essencial para o negócio no sentido de sobressair-se em relação à competição. É um atributo que garante a sobrevivência na indústria de software (SOWUNMI; MISRA, 2015). Portanto, é importante que o gerente de projetos defina um plano de ação.

O plano de gestão da qualidade é o processo de identificar os requisitos ou os padrões de qualidade do projeto e suas entregas e de como o projeto demonstrará conformidade com os requisitos de qualidade relevantes. Além disso, envolve o fornecimento de orientações sobre como e quando a qualidade será gerenciada e validada ao longo do projeto (PMI, 2013).

A qualidade segundo Rios et al. (2012) tem duas definições de trabalho:

- × do ponto de vista do produtor, a qualidade é o cumprimento dos requisitos;
- × do ponto de vista do consumidor a qualidade é o atendimento às necessidades do cliente.

Pressman (2011) diz que a qualidade é “uma gestão de qualidade efetiva aplicada de modo a criar um produto útil que forneça valor mensurável para aqueles que o produzem e para aqueles que o utilizam”. Nessa definição, o autor enfatiza três pontos principais, descritos a seguir.

- ✖ **Gestão de qualidade efetiva:** envolve a definição de uma infraestrutura adequada para construir um produto de software de alta qualidade, que considera as práticas de engenharia de software, e atividades de apoio, como configuração de mudanças e atividades relacionadas ao controle da qualidade (revisões e inspeções).
- ✖ **Produto útil:** atende às necessidades dos usuários com confiabilidade. Além disso, satisfaz as necessidades explícitas (requisitos especificados) e implícitas (por exemplo, facilidade de uso, segurança, etc.) dos usuários.
- ✖ **Agregação de valor:** a qualidade agrupa valor ao fabricante no sentido de redução das atividades de manutenção referentes à correção de erros. Em relação ao usuário final, o ganho é no sentido da agilização de um processo de negócio.

Na engenharia de software, existem métodos preventivos e métodos de detecção que podem ser utilizados para o alcance da qualidade de software.

Um método preventivo é a prática da Garantia da Qualidade de Software (GQS), um processo bem definido e repetitivo integrado com o gerenciamento de projetos e os ciclos de vida do desenvolvimento de software para rever mecanismos de controle interno e garantir a aderência aos padrões e procedimentos de software. O objetivo desse processo é assegurar a conformidade com os requisitos, reduzir o risco, avaliar os controles internos e melhorar a qualidade, o calendário estabelecido e as restrições orçamentais (SOWUNMI; MISRA, 2015).

Um método de detecção é a prática de Controle de Qualidade de Software (CQS), um processo pelo qual a qualidade do produto é comparada com os padrões aplicáveis; quando uma não conformidade é detectada, são tomadas as devidas providências (RIOS et al., 2012). Esse processo visa identificar os defeitos e corrigi-los.

Embora essas abordagens se complementem, existem diferenças em seu foco de atuação, como apresentado na figura 8.7.

Figura 8.7 – Diferenças entre GQS e CQS

	Controle de qualidade	Atua no Ciclo PDCA
		Garantia da qualidade
1.	Relacionado a um produto ou serviço específico.	1. Estabelece processo.
2.	Identifica defeitos com objetivo de corrigi-los.	2. Determina programas de medidas.
3.	Responsabilidade da equipe/funcionário.	3. Responsabilidade do gerenciamento.
		4. Avalia a eficiência do controle de qualidade.

Fonte: Rios et al. (2012).

A engenharia de software propõe essas técnicas, no entanto cabe ao gerente de projeto definir um plano de qualidade específico para um projeto particular, considerando as políticas de qualidade da organização. Isso implica uma seleção de padrões organizacionais específicos do projeto de software em questão e do processo de desenvolvimento a ser utilizado. Também especifica um processo de avaliação da qualidade que será realizado e define métricas de qualidade.

8.3.6 Gerenciamento do plano de projeto

Uma vez definido e aprovado o plano de projeto, o gerente de projeto deve realizar o gerenciamento de acordo com que ficou definido, como também atualizar o plano ao longo do projeto devido à identificação de mudanças no escopo.

Segundo o SWEBOK (IEEE, 2004), para projetos de software, em que a mudança é uma expectativa, os planos devem ser gerenciados. Portanto, a gestão do plano de projeto deve ser planejada. Planos e processos selecionados para desenvolvimento de software devem ser sistematicamente monitorados,

revisados, relatados e, quando apropriado, revisados. Planos associados a processos de apoio (por exemplo, documentação, gerenciamento de configuração de software e resolução de problemas) também devem ser gerenciados.

8.4 Formalização do plano de projeto

O projeto é iniciado, e suas atividades realizadas de acordo com o cronograma. No processo, recursos são utilizados (por exemplo, esforço pessoal, financiamento) e entregas são produzidas (por exemplo, documentos de projeto, como arquitetura do software, casos de testes e outros artefatos) (IEEE, 2004).

O gerente de projeto deve formalizar a todos os envolvidos o plano de projeto. É importante que todos os envolvidos tenham conhecimento das atividades necessárias para realizar as entregas do projeto e enfatizar que o cronômetro está correndo.

Durante a execução do projeto, o gerente de projetos tem de seguir as atividades que foram definidas no plano de gerenciamento do projeto, a fim de atender às especificações do projeto. Isto envolve (PMI, 2013):

- ✖ coordenar pessoas e recursos, gerenciar as expectativas das partes interessadas e também integrar e executar as atividades do projeto, em conformidade com o plano de gerenciamento;
- ✖ realizar atualizações no planejamento devido às mudanças na linha de base.

Significa que a execução deve seguir a programação da maneira como foi planejada e nos prazos acordados entre as partes interessadas. Quanto maior ou mais complexo o projeto, maior a necessidade de coordenação, comunicação e controle por parte do gerente de projeto para realizar as atividades de integração relacionadas às atividades que foram definidas no plano de projeto.

A integração entre os planos e a avaliação contínua de cada tarefa é uma das obrigações do gerente de projetos. Por exemplo, as entregas são avaliadas em termos de suas características a partir das atividades definidas no plano de qualidade para garantir que o projeto empregará todos os processos necessários para atender aos requisitos do projeto. Esse processo é executado em conjunto com o monitoramento e controle para saber se:

- × a equipe do projeto está seguindo os procedimentos definidos no plano de qualidade?
- × os padrões estão sendo atendidos?
- × o trabalho pode ser melhorado?

Normalmente, esse processo é executado por grupo específico, como o grupo de garantia da qualidade.

O gerente de projetos obtém as pessoas com o perfil profissional esperado e também as auxilia no aperfeiçoamento das competências técnicas, conforme o planejado no plano de recursos humanos. Os procedimentos de comunicação definidos são aplicados para que as informações cheguem às partes interessadas do projeto no formato e no período correto. Os riscos do projeto, bem como os custos e o cronograma das entregas são avaliados.

Há projetos que precisam, para sua execução, ser completamente ou parcialmente executados por fornecedores ou terceiros. Para isso, é necessária a definição de contratos, bem como seu monitoramento para acompanhar o desempenho dos fornecedores e o aceite final quando à conclusão da entrega de seus produtos (IEEE, 2004).

Além disso, o SWEBOK (IEEE, 2004) chama a atenção para as atividades de controle de processo. Nesse sentido, os resultados das atividades de monitoramento do projeto fornecem a base na qual as decisões podem ser tomadas. Por exemplo, quando a probabilidade e o impacto dos fatores de risco forem compreendidos, as mudanças podem ser feitas ao projeto. Isso pode assumir a forma de ação corretiva (por exemplo, testar novamente determinados componentes de software). Pode envolver a incorporação de ações adicionais (por exemplo, decidir usar a prototipagem para auxiliar na validação de requisitos de software). Conforme já mencionado, pode também implicar uma revisão do plano do projeto e outros documentos do projeto (por exemplo, a especificação de requisitos de software) para acomodar eventos inesperados e suas implicações.

8.5 Revisão e avaliação

Uma importante atividade no gerenciamento de projetos refere-se à avaliação de desempenho ou do andamento do projeto. Assim, além de avaliar a

situação atual do projeto, é preciso também verificar se as tendências apontam para um bom resultado. Configurando uma das funções principal do Gerente de Projetos, que é garantir que os objetivos do projeto sejam atingidos.

Nesse sentido, o SWEBOK chama a atenção para dois tópicos: identificar a satisfação dos requisitos e avaliar o desempenho do projeto (“reportar status” ou relatório de status do projeto). Então, ao longo da execução do projeto, é importante avaliar o progresso em relação ao alcance dos objetivos declarados e a satisfação dos requisitos das partes interessadas (usuários e cliente). Similarmente, a realização de avaliações da eficácia do processo de software, equipe, ferramentas e métodos utilizados (IEEE, 2004).

Normalmente, para acompanhar se o projeto está atingindo os objetivos são realizadas avaliações ao longo do projeto a fim de medir seu desempenho.

A maioria das ações de monitoramento e controle de um projeto ocorre nas reuniões periódicas de acompanhamento e também nas reuniões dos marcos de projeto. Geralmente, nas reuniões é apresentado um relatório de desempenho do projeto que contém a comparação entre o planejado e o executado, com o objetivo de determinar se o custo (orçamento), o cronograma e o trabalho realizado (escopo), estão conforme o planejado e apontar eventuais desvios. Outras informações possíveis são riscos, utilização dos recursos humanos, previsões de prazo e custos, mudanças solicitadas, ajustes, ações corretivas recomendadas etc.

As revisões periódicas de desempenho para a equipe do projeto envolvem a discussão dos seguintes tópicos:

- ✖ apresentar informações do progresso do projeto;
- ✖ levantar e discutir os problemas identificados (prazo, escopo, custo, qualidade, configuração e conflitos entre os membros da equipe etc.);
- ✖ revisar os riscos do projeto;
- ✖ revisar responsabilidades das partes envolvidas;
- ✖ verificar o andamento das mudanças internas e externas;
- ✖ assegurar que os produtos/resultados (“entregas”) do projeto estejam em conformidade com o padrão de qualidade ou critérios de aceitação do produto.

As revisões de marcos de projeto podem servir como ponto de monitoramento e controle, mas não devem ser exclusivas. “Marcos” de projeto são um momento significativo ou evento no projeto (PMI, 2013) – por exemplo, a data de conclusão de um entregável (*deliverable*), que pode ser a conclusão de uma funcionalidade ou a conclusão de uma fase do projeto, entre outros. É um momento de reflexão sobre o desempenho do projeto e pode resultar em sua continuidade.

8.6 Fechamento

O encerramento de um projeto consiste na execução dos processos para finalizar todas as atividades de todos os grupos de processos de gerenciamento do projeto, visando concluir formalmente o projeto, a fase ou as obrigações contratuais (PMI, 2013).

Nesse estágio, o critério para o sucesso do projeto é revisto. Uma vez que o fechamento/término seja estabelecido, atividades *post mortem*, de melhoria do processo e de arquivamento são realizadas (IEEE, 2004).

O gerente de projetos deve verificar se as tarefas especificadas nos planos foram completadas de acordo com o especificado; se todos os produtos planejados foram entregues com as características aceitáveis; se os requisitos foram confirmados como satisfeitos e os objetivos do projeto foram alcançados. Esses processos normalmente envolvem todos os *stakeholders* e resultam na documentação de aceitação do cliente e quaisquer relatos de problemas remanescentes reconhecidos.

Síntese

Este capítulo apresentou a importância do gerenciamento de projetos na engenharia de software e sua relação com o processo de desenvolvimento. Gerenciar um projeto é uma tarefa complexa, pois envolve conhecimento técnico de determinado seguimento. Tal dificuldade ocorre especialmente nos projetos de software devido às próprias características do software, como intangibilidade e especificidade de cada projeto, que dificultam nas tarefas de planejamento e acompanhamento.

Devido às características dos produtos e serviços de software, é importante o processo de elicitação de requisitos. O gerente de projeto deve ter habilidades técnicas e competências pessoais para integrar todas as partes interessadas do projeto e, ao mesmo tempo, ter uma visão global e detalhada das atividades do projeto.

A gestão de pessoas é um fator crítico de sucesso do projeto. Não é uma tarefa trivial organizar uma equipe com capacidade técnicas e ao mesmo tempo com capacidade de interagir com a especificidade de cada profissional. Por isso, cada vez mais o mercado valoriza profissionais com capacidade de interação com o indivíduo.

Ao longo do projeto, a equipe deve identificar e planejar respostas para os riscos. A qualidade dos produtos de projeto deve ser planejada e monitorada e depende das políticas organizacionais.

Atividades

1. Explique como as características dos projetos de software podem gerar problemas para o gerenciamento de projetos de software.
2. Explique porque em projetos de software os gerentes de projeto têm dificuldades na gestão de pessoas.
 - a) As maiores falhas de projetos ocorrem por erros de execução, sem relação com o escopo.
 - b) O erro cometido na definição do escopo tem o mesmo valor que o cometido após a implementação do projeto.
 - c) Erros de escopo são raros.
 - d) Erros de escopo são os mais onerosos.
 - e) Erros no escopo não são causas relevantes no custo da execução do projeto.
4. Explique a diferença entre as práticas Garantia da Qualidade e Controle de Qualidade.

9

Ferramentas, Métodos e Técnicas na Engenharia de Software

INDEPENDENTEMENTE DO PROCESSO de desenvolvimento, a análise e o design, a implementação, a entrega e a manutenção do software são atividades complexas que precisam ser melhoradas e continuam a ser um desafio para a Engenharia de Software. Ao longo dos anos, essa engenharia vem oferecendo inúmeras soluções que incluem propostas de processos, métodos e ferramentas. Este capítulo tem como objetivo apresentar os conceitos sobre as ferramentas e os métodos da Engenharia de Software em seus aspectos fundamentais, apresentar os diferentes tipos de ferramentas de Engenharia de Software e apresentar as diferentes abordagens dos métodos de Engenharia de Software.

9.1 Introdução

Desde a conferência NATO Software Engineering Conferences, em 1968, a Engenharia de Software evoluiu em diversos sentidos (ROCHA; MALDONADO; WEBER, 2001), estruturando métodos, técnicas e ferramentas com o objetivo de aumentar a qualidade e a produtividade no processo de desenvolvimento.

Pressman (2011) explica a importância desses elementos no desenvolvimento de software:

- × **ferramentas da Engenharia de Software** – fornecem apoio automatizado ou semiautomatizado para o processo e para os métodos;
- × **métodos da Engenharia de Software** – fornecem as técnicas para a construção do software;
- × **processo da Engenharia de Software** – define a sequência de aplicação dos métodos técnicos, cria produtos (modelos, documentos, dados, formulários etc.), estabelece marcos, garante a qualidade e faz a gestão adequada das mudanças.

Portanto, constitui a base para o controle do gerenciamento de projetos de software. Em outras palavras, podemos dizer que os processos são os caminhos utilizados para desenvolver o projeto de software; já o método é a aplicação das tarefas a serem realizadas nas etapas do processo e as ferramentas são os recursos utilizados no método e no processo.

O processo funciona como uma base para a Engenharia de Software e a partir dele se define uma metodologia para a entrega efetiva de tecnologia e definem-se os métodos que auxiliam na definição e na gestão do processo de desenvolvimento. As ferramentas de apoio ao desenvolvimento de software (Computer-Aided Software Engineering – CASE) representam uma parte importante das tecnologias de apoio utilizadas para desenvolver e manter sistemas de Tecnologia da Informação.

9.2 Ferramentas de Engenharia de Software

O projeto, a implementação, a entrega e a manutenção do software são atividades complexas e dispendiosas que precisam ter melhor controle, e uma

das alternativas é a automação das atividades de desenvolvimento de software (FUGGETTA, 1993). As ferramentas de Engenharia de Software, desde sua criação, têm se tornado uma proposta viável para empresas que almejam obter sucesso em um mercado cada vez mais exigente com relação a prazos devido à promessa de redução de tempo nas atividades de desenvolvimento de software.

No início de sua concepção, as ferramentas de Engenharia de Software eram consideradas como a “bala de prata” do desenvolvimento das aplicações por suas promessas de automatização das atividades (MCMURTREY et al., 2000). No entanto, após anos de pesquisas e com sua aplicação no mercado, algumas limitações foram verificadas, principalmente em relação à integração de novos componentes e devido às ferramentas serem de uso geral, ou seja, não serem específicas ao domínio da aplicação (PRESSMAN, 2011).

Embora essa solução não tenha atendido a todas as expectativas, esses produtos continuam sendo uma opção viável para os desenvolvedores de aplicativos modernos (MCMURTREY et al., 2000).

9.3 Definições e objetivos

A norma ISO/IEC 14102 (INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, 2008) define ferramentas CASE como ferramentas de software que podem auxiliar engenheiros de software fornecendo apoio automatizado para atividades do ciclo de vida de software. Pressman (2011) define as ferramentas integradas como Engenharia de Software com auxílio do computador, de modo que as informações criadas por uma ferramenta possam ser usadas por outras; um aplicativo que auxilia os profissionais envolvidos no desenvolvimento de software.

São ferramentas automatizadas que têm como objetivo auxiliar o desenvolvedor de sistemas em uma ou várias etapas do ciclo de criação de software. As ferramentas CASE podem ser utilizadas em vários modos (INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, 2008):

- × ferramentas autônomas – apenas a compatibilidade com elementos do ambiente deve ser abordada;
- × pequenos grupos que se comunicam diretamente uns com os outros – pode-se supor que a integração é predefinida, talvez proprietariamente;

- × grande escala – SEE (Software Engineering Environment) – a capacidade da ferramenta para utilizar os serviços relevantes da estrutura deve ser abordada. Essa solução tem como premissa a existência de um conjunto integrado de ferramentas que agilizam e reduzem o esforço de trabalho, integrando e automatizando todas as fases do ciclo de vida do software.

Dentre os objetivos do uso dessas ferramentas destacam-se melhorar a qualidade do software, automatizar as tarefas repetitivas, aumentar e agilizar a produtividade do processo de desenvolvimento de software, gerar automaticamente a documentação do sistema e colaborar com a gestão dos artefatos produzidos.

9.4 Ferramentas CASE

O mercado de ferramentas CASE oferece centenas de soluções para aperfeiçoar e implementar cada vez mais recursos para atender às necessidades das atividades do ciclo de vida dos projetos de software (PRESSMAN, 2011), tornando possível agilizar o desenvolvimento desses projetos e gerando maior rapidez e melhores resultados. Ao longo dos anos, autores têm classificado as ferramentas CASE de diferentes maneiras. A classificação dada por Pressman (2011) baseia-se na identificação das diferentes funções suportadas pelos produtos CASE, como planejamento de sistemas de negócios, gerenciamento de projetos, suporte (documentação, banco de dados, gerenciamento de configuração etc.), análise e projeto, programação, integração e teste, prototipagem e manutenção.

O autor também destaca ferramentas de código aberto, a maioria específica para as atividades de programação, e, em nível industrial, as ferramentas de IDE (Integrated Development Environment ou Ambiente de Desenvolvimento Integrado, em português), que integram um conjunto de ferramentas individuais ao redor de uma base de dados central (repositório). Essas ferramentas integram as informações de um processo de software que podem auxiliar na colaboração de muitos sistemas grandes e complexos.

No entanto, existem algumas limitações relacionadas a esses ambientes integrados:

- ✖ dificuldade de integração de novas ferramentas que não pertencem ao pacote inicial;
- ✖ tendência a serem de uso geral, ou seja, não são específicas ao domínio da aplicação;
- ✖ demanda de tempo para introdução de novas soluções de tecnologia, por exemplo, desenvolvimento de software controlado por modelo;
- ✖ disponibilidade de ambiente de Engenharia de Software (SEE) que suporte a nova tecnologia.

Outra classificação é apresentada por McMurtrey et al. (2000), que categoriza as ferramentas de Engenharia de Software de acordo com as etapas do ciclo de desenvolvimento de um projeto de software:

- ✖ Front end ou Upper CASE – ferramentas que apoiam as etapas iniciais de criação dos sistemas, como as fases de planejamento, análise e projeto do software; também têm a possibilidade de gerar protótipos, permitindo que os próprios analistas criem as simulações do sistema;
- ✖ Back end ou Lower CASE – ferramentas que dão apoio à parte física, isto é, à codificação, aos testes e à manutenção da aplicação;
- ✖ I-CASE ou Integrated CASE – ferramentas que cobrem todo o ciclo de desenvolvimento de software, desde a fase de especificação de projeto e análise de requisitos até o controle final da qualidade do software implementado.

As próximas seções apresentam em mais detalhes algumas das funcionalidades das ferramentas CASE presentes no mercado e em uso pelos profissionais de desenvolvimento de software.

9.4.1 Classificação por funcionalidades

O quadro 9.1 ilustra, em termos gerais, algumas funcionalidades das ferramentas, conforme Fuggetta (1993). Por exemplo as ferramentas de planejamento, edição de texto, preparação de documentos, gestão de configuração, que podem ser usadas em todas as fases do ciclo de vida do projeto de software.

Quadro 9.1 - Classe e subclasse de ferramentas CASE

Classe	Subclasse
Edição	Editor gráfico Editor de texto
Programação	Codificação e debugs Assembles Compiladores Interpretadores Pré-compiladores e processadores Geradores de código Geradores de compilador Reestruturadores do código
Verificação e validação	Analisadores estáticos Geradores de referência cruzada Fluxogramas Verificadores de sintaxe Analisadores dinâmicos Instrumentos de programação Comparadores Executores simbólicos Emuladores Assistentes de correção Geradores de casos de teste Ferramentas de gerenciamento de testes

Classe	Subclasse
Gerenciamento de configuração	Gerenciadores de versão e configuração Monitores de controle de mudança Bibliotecas
Métricas e medição	Analisadores de código Monitores de execução/analisadores de tempo
Gerenciamento de projetos	Ferramentas de estimativa de custos Ferramentas de planejamento de projetos Quadros de avisos Portáteis do projeto
Ferramentas diversas	Sistemas de hipertexto Planilhas

Fonte: Fuggetta (1993).

9.4.1.1 Ferramentas de edição

As ferramentas de edição (editores) podem ser classificadas em duas subclasses: editores de texto e editores gráficos. A primeira subclass inclui editores de texto tradicionais e processadores de texto usados para produzir documentos textuais, tais como programas, especificações textuais e documentação. A segunda subclass é usada para produzir documentos usando símbolos gráficos.

Exemplos: Microsoft Word, Javadoc, Wiki, Open Office, Eclipse, NetBeans, Astah e ferramentas para inserir especificações gráficas (como aquelas baseadas em diagramas de fluxo de dados).

9.4.1.2 Ferramenta de programação

Essas ferramentas são usadas para apoiar a codificação e a reestruturação de código (manutenção). As duas principais subclasses são as ferramentas de codificação e depuração e as de geração de código-fonte.

A primeira subclasse inclui ferramentas tradicionais usadas para compilar a execução e depurar um programa. Exemplos: compiladores – JDK, compiladores C baseados em Unix. A segunda classe inclui ferramentas que geram código. Um gerador de código é desenvolvido para criar automaticamente um código-fonte de alto nível em linguagens de programação. Exemplos: .NET, C++, C#, Java e outros.

9.4.1.3 Ferramentas de verificação e validação

Essa classe inclui ferramentas que suportam validação e verificação de programas. A validação visa garantir que as funções do produto são o que o cliente realmente deseja, ao passo que a verificação visa garantir que o produto em construção atenda à definição de requisitos (SOMMERVILLE, 2011). Essa classe tem muitas subclasses (FUGGETTA, 1993):

- ✗ a análise estática examina um programa de computador sem executar o programa (estático) e a análise dinâmica examina ou monitora a execução do programa (dinâmico);
- ✗ os comparadores equiparam dois arquivos para identificar pontos em comum e diferenças; normalmente, são usados para comparar os resultados do teste com o programa esperado nas saídas;
- ✗ os emuladores/simuladores imitam todo um sistema de computador ou parte dele e aceitam os mesmos dados, fornecem as mesmas funcionalidades e obtêm os mesmos resultados que o sistema imitado;
- ✗ os assistentes de prova de correção suportam técnicas formais para provar matematicamente que um programa satisfaz suas especificações ou que uma especificação satisfaz determinadas propriedades;
- ✗ a geração de teste-case toma como entrada um programa de computador e uma seleção de critérios de teste e gera dados de entrada de teste que atendam a esses critérios;
- ✗ as ferramentas do gerenciador de teste suportam o teste gerenciando os resultados desses testes: listas de verificação de teste, testes de regressão, teste de métricas de cobertura e assim por diante.

Exemplos: ferramentas de testes unitários: JUnit, cunit (unitários), comparador de resultados de teste: HP Basic Branch Analyzer, verificador de Sintaxe SQL e verificação de sintaxe de código PHP – PiliApp e outros.

9.4.1.4 Ferramentas de gerenciamento de configuração

As técnicas de gestão da configuração coordenam e controlam a construção de um sistema composto por várias partes. O desenvolvimento e o gerenciamento de software podem se beneficiar grandemente do gerenciamento de configuração, que pode ser decomposto nas seguintes tarefas (FUGGETTA, 1993):

- × **gerenciamento de versões** – durante o desenvolvimento do software, é produzida mais de uma versão de cada item de software e essas versões devem ser gerenciadas para que o trabalho subsequente incorpore a versão correta;
- × **identificação do item** – cada item de software deve ser inequivocamente identificável e os agentes de processo de software (todas as pessoas que trabalham no processo de software) devem ser capazes de recuperar itens específicos para construir e reconstruir configurações coerentes do produto em desenvolvimento;
- × **construção de configuração** – um produto de software é uma coleção complexa de itens de software versionados e a construção de um produto requer operações, como pré-processamento, compilação e ligação a um grande conjunto de itens de software;
- × **alteração de controle** – as alterações a um item de software podem ter impacto em outros componentes, além disso, se vários programadores podem acessar os mesmos itens de software, o controle é necessário para sincronizar as atividade e evitar a criação de versões inconsistentes ou errôneas;
- × **gerenciamento de biblioteca** – todos os itens de software relevantes em um processo de software devem estar sujeitos a políticas efetivas de armazenamento e recuperação.

Exemplos: ferramentas de controle de versões: CVS, Subversion, IBM Rational ClearCase e Microsoft Visual Source Safe; ferramentas de controle de modificações: Bugzilla, Jira, e IBM Rational ClearQuest.

9.4.1.5 Ferramentas de medição e métricas

Ferramentas que coletam dados sobre programas e execução de programas se dividem em duas subclasses:

- × ferramentas para analisar o código-fonte e calcular várias métricas de código-fonte (para avaliar a complexidade do código de acordo como as métricas de McCabe, por exemplo);
- × ferramentas para monitorar a execução de programas e coletar estatísticas de tempo de execução.

Exemplos: Costar; Calico; USC-COCOMO; Complexity Tool.

9.4.1.6 Ferramentas de gerenciamento de projetos

Os gerentes de projetos contam com um amplo conjunto de ferramentas de gerência que, quando utilizadas, trazem significativas melhorias na qualidade do trabalho realizado no esforço de desenvolvimento de software tanto para pequenos quanto para grandes projetos. Segundo Pressman (2011), ao usar um conjunto selecionado de ferramentas CASE, o gerente de projetos pode gerar estimativas de esforço, custo e duração de um projeto de software, definir uma estrutura de divisão de trabalho, planejar uma programação viável de projeto e acompanhar projetos em base contínua.

Com a utilização de ferramentas de gerência, pode-se obter métricas que demonstrarão índices de produtividade de desenvolvimento de software e níveis de qualidade do produto. Vários tipos de produtos suportam gerenciamento de projetos. Nessa categoria Fuggetta (1993) destaca três subclasses.

A primeira subclass inclui produtos usados para estimar custos de produção de software e geralmente implementam técnicas como o Cocomo (modelo de custo construtivo) ou pontos de função e fornecem interfaces de fácil utilização para especificar as informações do projeto e analisar os resultados da estimativa.

A segunda subclass compreende ferramentas que suportam o planejamento do projeto, isto é, programação de projetos, atribuição de recursos e acompanhamento de projetos, e é baseada em conceitos bem conhecidos e notações como WBS (Estrutura Analítica de Projetos – EAP – do inglês Work

Breakdown Structure) e gráficos de Gantt (avaliação do programa e técnica de revisão).

A terceira subclasse inclui ferramentas para apoiar a comunicação e a coordenação entre os membros da equipe do projeto. Algumas permitem a interação on-line entre as pessoas, por exemplo, sistemas de teleconferência (também chamadas de mesas de conferência), sistemas de e-mail e quadros de avisos eletrônicos. Outras ferramentas são agendas de projetos usadas para coordenar atividades e reuniões, como Microsoft Project, dotProject, Xplaner. Também fazem parte dessa categoria as ferramentas de rastreamento de requisitos, que têm por finalidade corrigir o problema da diferença entre o sistema entregue e os requisitos originalmente definidos pelo usuário.

Segundo Pressman (2011), o objetivo das ferramentas de rastreamento de requisitos é oferecer uma abordagem sistemática dos requisitos, que se iniciam com a especificação do cliente. Ferramentas típicas dessa categoria combinam a avaliação interativa de textos de usuários com um sistema gerenciador de banco de dados que tem por função armazenar e categorizar cada requisito do sistema a partir da especificação original.

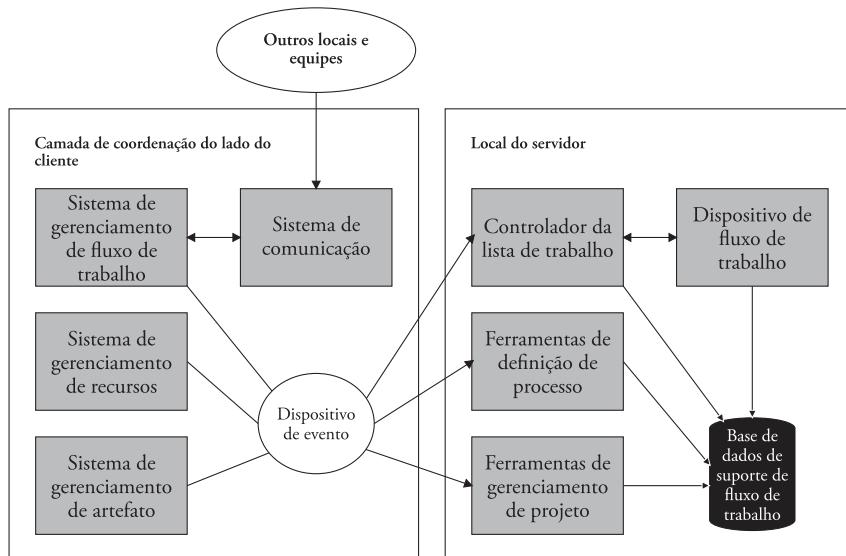
9.5 Ferramentas de ambientes integrados colaborativos

Conforme exposto na seção anterior, existe uma variedade de ferramentas desenvolvidas para atender às necessidades dos profissionais que desenvolvem software. Mas, com a inserção das metodologias ágeis, a nova forma de trabalhar com equipes globais fez que surgisse a necessidade de desenvolver novas ferramentas.

Pressman (2011) cita duas tendências na automação de ferramentas de Engenharia de Software: ferramentas que respondem a tendências leves e ferramentas que lidam com as tendências tecnológicas. A primeira tendência corresponde às necessidades de gerenciar e controlar os requisitos emergentes, estabelecer modelos de processos que acomodem as mudanças e apoiar a coordenação de equipes distribuídas. As pesquisas nesse campo sugerem a criação de ambientes de Engenharia de Software (SEE) colaborativo.

A figura 9.1 apresenta um exemplo de pesquisa sobre um ambiente denominado GENESIS, uma iniciativa de código aberto com foco em suportar trabalho colaborativo em Engenharia de Software.

Figura 9.1 - Arquitetura SEE colaborativa



Fonte: Adaptada de Pressman (2011) apud Aversan (2004).

A figura mostra dois subsistemas. O primeiro é composto por um sistema de gerenciamento de fluxo de trabalho (gerencia as atividades), um sistema de gerenciamento de recursos (aloca recursos para os diferentes projetos), um sistema de gerenciamento de artefato (gerencia os diferentes artefatos do projeto) e um sistema de comunicação (suporta a comunicação entre as equipes distribuídas). O dispositivo de eventos coleta os eventos que ocorrem durante o processo e notifica os outros (por exemplo, o término de um teste).

O segundo subsistema é composto por um dispositivo de fluxo de trabalho (interage com o dispositivo de trabalho para propagar eventos), uma ferramenta de definição de processo (permite à equipe definir novas atividades), uma ferramenta de gerenciamento de projeto (permite criar e gerenciar planos de projeto) e um controlador da lista de trabalho (fornecendo informações sobre as tarefas do projeto).

A segunda tendência refere-se à criação de um conjunto de ferramentas que suporta desenvolvimento controlado por modelo com a ênfase no projeto controlado por arquitetura, com foco no modelo, não no código-fonte.

9.6 Método de Engenharia de Software

Os modelos e métodos de Engenharia de Software foram desenvolvidos com o objetivo de tornar essa atividade sistemática, repetitiva e, em última instância, mais orientada para o sucesso (INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, 2004). Essas abordagens variam amplamente em escopo, desde endereçar uma única fase do ciclo de vida do software até cobrir o ciclo de vida completo dele. Existem vários métodos que as organizações podem escolher, no entanto, é importante para o engenheiro de software escolher um método adequado ou métodos para a tarefa de desenvolvimento de software (ABRAN; DUPUIS, 2004). Faz-se necessário, portanto, considerar o tipo de projeto, o negócio da organização e a equipe, entre outros aspectos, pois essa escolha pode ter efeito sobre o sucesso do projeto de software.

As próximas seções apresentam alguns métodos de Engenharia de Software. As áreas temáticas foram organizadas em discussões de métodos heurísticos, métodos formais, métodos de prototipagem e métodos ágeis, de acordo com a visão do *SWEBOK* (ABRAN; DUPUIS, 2004).

9.6.1 Métodos heurísticos

Os métodos heurísticos são baseados na experiência de suas práticas na indústria de software. Esse tópico contém três categorias gerais (ABRAN; DUPUIS, 2004): análise estruturada, modelagem de dados e análise orientada a objetos.

A primeira categoria é a análise estruturada, que compreende modelos de software desenvolvidos a partir do ponto de vista funcional (requisitos do sistema) e comportamental (definição de processos internos do sistema), de uma visão de alto nível do software (incluindo dados e elementos de controle) e, em seguida, decompondo ou refinando progressivamente os componentes do modelo por meio de diagramas cada vez mais detalhados. Exemplos

de técnicas de análise estruturada são análise estruturada e análise essencial (YOURDON, 1992). A análise estruturada e a essencial são técnicas para especificação de sistemas que modelam os dados e as funções de um sistema. Só que a análise essencial incorporou o conceito de “eventos” como a principal ferramenta para o particionamento funcional do sistema. O Diagrama de Fluxo de Dados (DFD) é uma representação dos processos do sistema que apresentam partes dos componentes do sistema e as interfaces entre eles. O Dicionário de Dados é um conjunto organizado das definições lógicas de todos os nomes de dados mostrados no DFD.

A segunda categoria é a modelagem de dados. O modelo de dados é construído do ponto de vista dos dados ou das informações utilizadas, no qual as tabelas de dados e as relações definem os modelos de dados. Esse método é usado principalmente para definir e analisar requisitos de dados que suportem projetos de banco de dados ou repositórios de dados normalmente encontrados em software de negócios, em que os dados são gerenciados ativamente como um recurso ou recurso de sistemas de negócios. Exemplos de modelos de análise nessa categoria são:

- × **modelagem conceitual** – usada como representação em alto nível para envolver o cliente, pois nessa etapa visa discutir os aspectos do negócio do cliente, não da tecnologia;
- × **modelagem lógica** – com base no modelo conceitual, agrupa mais alguns detalhes de implementação (chaves primárias, estrangeira, normalização, entre outras);
- × **modelagem física** – com base no modelo lógico, demonstra como os dados são fisicamente armazenados no banco de dados.

A última categoria é a análise orientada a objetos, a mais nova abordagem de análise de sistemas, baseada na decomposição do sistema de acordo com os objetos que serão manipulados por ele. Oferece como principais benefícios uma visão do sistema mais próxima do mundo real, uma modelagem do sistema baseada nos dados e uma maior transparência da análise para o projeto.

É uma forma de desenvolvimento de sistemas de software que o trata como um conjunto de componentes que interagem entre si para resolver um problema. Esses componentes são denominados objetos e permitem ser

combinados ou utilizados separadamente, pois apresentam baixa dependência externa e alta coesão interna, o que possibilita promover substituições ou realizar manutenções sem afetar os demais componentes (FURLAN, 1998).

A seguir, alguns principais conceitos relacionados à orientação a objetos, segundo Guedes (2011):

- ✖ objetos são uma ocorrência específica de uma classe, ou seja, uma “instância de classe”;
- ✖ método é a lógica contida em uma classe para atribuir-lhe comportamentos; são as funções e os procedimentos contidos na classe;
- ✖ objetos se comunicam por meio de mensagens, que são trocas de informações ou requisições através da execução de métodos, operações ou comportamentos dos objetos;
- ✖ visibilidade indica a acessibilidade do objeto (pública, privada, protegida);
- ✖ herança permite o reaproveitamento de atributos e métodos dos objetos por meio do conceito de classes e subclasses.

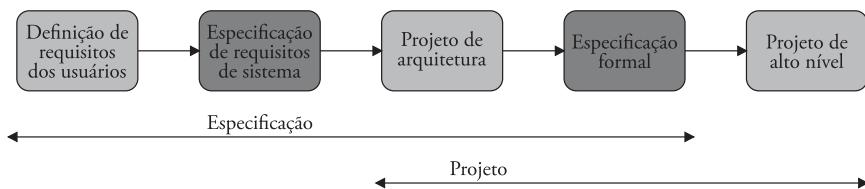
O tipo de notação desse modelo de análise é Unified Modeling Language (UML, Linguagem de Modelagem Unificada, em português), uma linguagem de modelagem que permite representar um sistema de forma padronizada (GUEDES, 2011).

9.6.2 Métodos formais

Os métodos formais são baseados em formalismos matemáticos para especificação, desenvolvimento e verificação dos sistemas de softwares e hardware (SOMMERVILLE, 2011). A fase de especificação formal é uma descrição inequívoca do que o sistema deve fazer. Usando métodos manuais ou apoiados por ferramentas, é possível verificar se o comportamento de um programa é consistente com a especificação (SOMMERVILLE, 2011).

As especificações formais são desenvolvidas como parte de um processo dirigido a planos, conforme ilustra a figura 9.2.

Figura 9.2 - Especificação formal de um processo baseado em planos



Fonte: Adaptado de Sommerville (2011).

Os requisitos e o projeto do sistema são especificados em detalhes e cuidadosamente analisados e verificados antes do início da implementação. A especificação formal consiste em formalizar os requisitos descobertos utilizando algum método formal, criando uma modelagem com o uso de elementos, mas a especificação formal ocorre antes de o projeto ser detalhado. Os benefícios dessa abordagem são:

- ✗ reduzir o número de erros do sistema;
- ✗ reduzir o tempo de desenvolvimento;
- ✗ proporcionar melhor documentação;
- ✗ melhorar a comunicação (entre equipe e usuário);
- ✗ facilitar a manutenção;
- ✗ ajudar a organizar as atividades durante o ciclo de vida.

Uma especificação formal utiliza uma linguagem formal que provê uma notação (domínio sintático), um universo de objetos (domínio semântico) e uma regra precisa que define que os objetos satisfazem cada especificação (PRESSMAN, 2011). O domínio sintático é definido em termos de um conjunto de símbolos (por exemplo, constantes, variáveis e conectores lógicos) e um conjunto de regras gramaticais para combinar esses símbolos (há necessidade de sentenças bem formadas). O domínio semântico indica como a linguagem representa os requisitos do sistema. São exemplos de linguagens formais de especificação: OCL; LARCH; VDM e teoria de conjuntos em notação B ou Z (PRESSMAN, 2011).

9.6.3 Métodos de prototipagem

Muitas vezes, clientes têm em mente um conjunto geral de objetivos para um sistema de software, mas não são capazes de identificar claramente as funcionalidades ou as informações (requisitos) que o sistema terá de prover ou tratar. A prototipação é uma técnica para ajudar analistas de software e clientes a entender o que está sendo construído quando os requisitos não estão claros (SOMMERVILLE, 2011). Um protótipo é uma representação limitada do design de um sistema, que pode ser um esboço de uma tela ou um conjunto de telas com os seguintes objetivos:

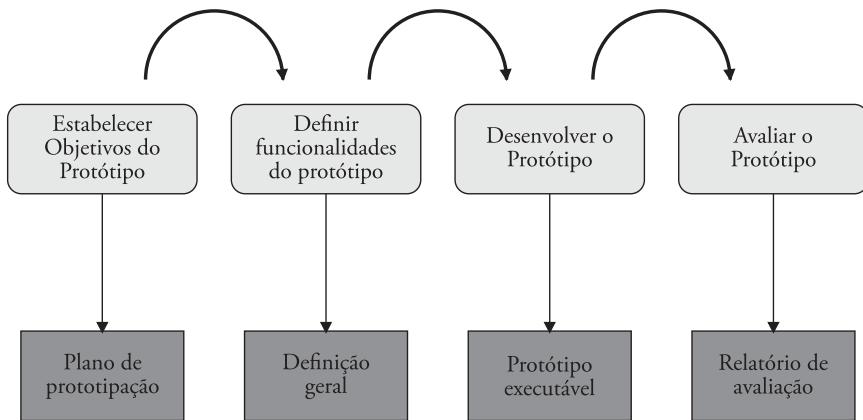
- ✖ entender os requisitos do usuário e obter uma melhor definição dos requisitos do sistema;
- ✖ possibilitar que o desenvolvedor crie um modelo (protótipo) do software que deve ser construído;
- ✖ possibilitar o entendimento da equipe sobre os requisitos do produto e tecnologias que não têm experiência de desenvolvimento.

Entre as vantagens, pode-se destacar:

- ✖ possibilidade de receber o feedback do cliente rapidamente (interação entre equipe do projeto e usuário) por meio da visualização do produto em estágios iniciais do projeto de software;
- ✖ tempo e custos reduzidos, pois os protótipos podem melhorar a qualidade de requisitos e de especificações fornecidas aos desenvolvedores, o que evita os custos de retrabalho, já que, após a versão final do software, qualquer tipo de mudança poderá aumentar exponencialmente o custo do software, além de estourar os prazos definidos no início.

Um exemplo de modelo de processo para desenvolvimento de protótipos é apresentado na figura 9.3.

Figura 9.3 - Processo de desenvolvimento de protótipo



Fonte: Sommerville (2011).

Os objetivos da prototipação devem ser previamente estabelecidos (validar interface de usuário ou requisitos, verificar viabilidade técnica, entre outros). Após definir o escopo das funcionalidades do protótipo, desenvolver o protótipo definido e, no estágio final, avaliar o protótipo (definir planos de avaliação com base nos objetivos estabelecidos).

Segundo o SWEBOK (ABRAN; DUPUIS, 2004), um protótipo pode ser usado ou avaliado de várias maneiras pelo engenheiro de software ou por outras partes interessadas no projeto, impulsionado principalmente pelas razões subjacentes que levaram ao desenvolvimento do protótipo, em primeiro lugar. Os protótipos podem ser avaliados ou testados contra o software realmente implementado ou contra um conjunto de requisitos-alvo (por exemplo, um protótipo de requisitos). O protótipo também pode servir como um modelo para um futuro esforço de desenvolvimento de software (por exemplo, em uma especificação de interface de usuário).

Os protótipos podem ser classificados como protótipos de baixa e de alta fidelidade (HAKIM; SPITZER, 2000). Segundo o autor, fidelidade refere-se ao grau em que o protótipo reflete a largura, a profundidade e o tamanho do produto pretendido. Os protótipos de baixa fidelidade retratam partes da aplicação pretendida, construídos sem muito esforço, geralmente em maque-

tes de papel ou cartão, de interfaces de usuário para revisão com pequenos grupos de usuários. Para minimizar o custo de produção desses protótipos de baixa fidelidade, raramente são incluídas capacidades interativas “embutidas”. Exemplos são apresentados na figura 9.4.

Figura 9.4 - Exemplo de protótipos de baixa fidelidade



Fonte: Oliveira (2009).

A alta fidelidade geralmente se refere a uma versão alfa do produto final (versões incompletas), construída usando ferramentas de produção, mas faltando a conclusão de certos recursos. São considerados protótipos porque são demonstrados aos clientes e usuários com antecedência suficiente no processo de desenvolvimento para fazer ajustes com base em sua entrada. Exemplos de ferramentas de protótipos funcionais são mockups ou protótipos funcionais.

9.6.4 Métodos ágeis

Os métodos ágeis são uma alternativa de desenvolvimento de software aos métodos dirigidos a planos cuja premissa é o mínimo de documentação e colaboração de uma equipe motivada e competente na atividade de desenvolvimento de software. No fim da década de 1980, houve uma ênfase muito forte na qualidade do processo, quando a melhor forma de desenvolver software era por meio de rigorosos planos de projeto apoiados por ferramentas CASE. Essa percepção veio das empresas que desenvolviam projetos de software com o escopo grande, como as empresas aeroespaciais e de governo (SOMMERVILLE, 2011). No entanto, ao longo dos anos foi-se percebendo que essas abordagens pesadas dirigidas a planos não eram eficazes para proje-

tos pequenos e com equipes pequenas (gastava-se mais tempo com análises do que com o desenvolvimento da aplicação).

Foi devido a essa constatação que surgiram as metodologias ágeis, a partir do manifesto ágil no início da década de 2000. A seguir, são descritos dois métodos ágeis segundo o *SWEBOK* (ABRAN; DUPUIS, 2004).

O XP (Extreme Programming) é uma abordagem que usa histórias ou cenários para requisitos, produz testes antes do desenvolvimento dos componentes, tem envolvimento direto do cliente na equipe (geralmente definindo testes de aceitação), usa programação em pares e fornece refatoração e integração de código contínuo. As histórias são decompostas em tarefas, priorizadas, estimadas, desenvolvidas e testadas. Cada incremento de software é testado com testes automatizados e manuais e um incremento pode ser liberado com frequência, a cada quatro semanas.

O Scrum é uma metodologia ágil para gestão e planejamento de projetos de software, no qual os projetos são divididos em ciclos (tipicamente mensais) chamados de Sprints. O Sprint representa um Time Box dentro do qual um conjunto de atividades deve ser executado. O Scrum Master gerencia as atividades dentro do incremento do projeto e cada incremento não dura mais de 30 dias. É desenvolvida uma lista de produtos (Backlog do produto) a partir da qual as tarefas são identificadas, definidas, priorizadas e estimadas. Uma versão de trabalho do software é testada e liberada em cada incremento e reuniões diárias são realizadas para acompanhar as atividades do projeto.

Existem novos métodos resultantes de combinações de métodos ágeis e baseados em planos, nos quais os profissionais estão definindo novos métodos que equilibram os recursos necessários tanto em métodos pesados quanto em métodos leves, baseados principalmente na estrutura organizacional predominante nas necessidades de negócios:

estas necessidades empresariais, como normalmente representadas por alguns dos stakeholders do projeto, devem direcionar a escolha na utilização de um método de engenharia de software sobre outro ou na construção de um novo método a partir das melhores características de uma combinação de métodos de engenharia de software. (ABRAN; DUPUIS, 2004)

Síntese

Este capítulo apresentou conceitos e tipos de ferramentas e métodos de Engenharia de Software cujas ferramentas são uma solução para diminuir o esforço e a complexidade das atividades do ciclo de vida do projeto. No entanto, devido ao surgimento de novos métodos de desenvolvimento ágil e também ao desenvolvimento de software por equipes distribuídas, surgiu a necessidade de desenvolver novas ferramentas CASE.

A atividade de análise e projeto de software têm evoluído ao longo dos anos de acordo com o paradigma de programação. A especificação com métodos formais, apesar de ser onerosa, tem a vantagem do entendimento detalhado do sistema. Os métodos ágeis são metodologias de desenvolvimento que têm como ponto forte a agilidade por meio da colaboração da equipe e do cliente.

Atividades

1. Explique o conceito de ferramentas CASE e cite os objetivos de seu uso.
2. Descreva a classificação das ferramentas CASE que consideram o ciclo de vida do projeto.
3. Explique por que surgiram as metodologias ágeis de desenvolvimento.
4. Na fase de desenvolvimento do Scrum, o software é desenvolvido em processos iterativos denominados:
 - a) Building Products.
 - b) Product Backlog.
 - c) Sprint.
 - d) Product Owner.
 - e) Product Backlog Cycle.

10

Qualidade de Software

O MERCADO DE software se tornou mais concorrido com o crescente aumento de empresas voltadas para o desenvolvimento de software. Em um mercado tão agressivo, não há como se tornar competitivo sem qualidade. As organizações de software buscam cada vez mais investir na qualidade de seus produtos e serviços, pois qualidade deixou de ser um diferencial e passou a ser um requisito básico.

NÃO BASTA A empresa dizer que tem qualidade, pois seus clientes terão padrões de qualidade e exigências que muitas vezes nem sabiam que teriam. Somente após conhecer algum software concorrente, os clientes compreendem que determinada característica do software lhes seria muito conveniente, facilitando seu trabalho. Qualidade de software é algo complexo por apresentar requisitos implícitos e subjetivos.

NESTE CAPÍTULO, É apresentada uma visão geral sobre a qualidade de software, fornecendo algumas perspectivas sobre qualidade de software e alguns modelos de gestão e de critérios para identificar um software de qualidade.

10.1 Fundamentação da qualidade de software

Qualidade é um conceito subjetivo. O que é visto como qualidade por um, não é o mesmo visto por outro, já que a questão da qualidade envolve também expectativas. Tanto é assim que a palavra “qualidade”, quando pesquisada, apresenta conceitos diversificados, cada qual com aspectos distintos, já que muda de pessoa para pessoa, de empresa para empresa e até de cultura para cultura. Isso porque qualidade não é um critério percebido sempre da mesma forma, não faz parte de um senso comum ou tem uma definição aceita por todos os profissionais ou segmentos de atuação das empresas. Qualidade, portanto, é vista e percebida de maneira muito singular.

Para melhor entendimento, imagine, por exemplo, o que seria para você um carro de qualidade: bastaria ter um motor potente com uma tecnologia avançada, ou teria também que ter um design diferenciado? Ou será que o ponto principal seria um carro com os melhores itens de segurança e que seja econômico? O que você entende como um carro de qualidade é similar ao que seu pai, amigos ou outras pessoas do seu convívio entendem? Será que uma fábrica americana de automóveis preconiza como qualidade as mesmas questões que uma fábrica japonesa? Ou chinesa? Ou brasileira?

A ênfase e a busca pela qualidade, seja no produto ou no serviço, ganham cada vez mais força e, ao mesmo tempo, o conceito também é aprimorado.

No passado, a revolução industrial propiciou a produção em escala dos produtos que antes, produzidos de modo artesanal, eram avaliados um a um pelo artesão conforme a exigência dos clientes. Já com a produção mecânica, os produtos eram feitos em grandes quantidades de maneira mecanizada, e assim não se conseguia garantir a qualidade da mesma forma que o artesão, surgindo com isso a inspeção como uma maneira de tentar garantir certa qualidade aos produtos. Além disso, começou a se desenvolver um fenômeno bastante comum para os dias atuais, mas ainda novo para a época: a concorrência.

Mesmo com a inspeção em cada etapa da produção, muitos produtos apresentavam defeitos, especialmente na Primeira Guerra Mundial, em relação aos produtos bélicos. Já na Segunda Guerra Mundial, a busca pela garantia da qualidade foi intensa, na qual o Japão acabou ganhando destaque. A partir disso, começaram a surgir modelos de Gestão da Qualidade. Isso fez

com que muitas empresas se voltassem ao tema de modo mais intensificado, buscando o entendimento sobre o assunto, especialmente do ponto de vista dos seus clientes. Com a concorrência cada vez maior, houve necessidade também da empresa se manter competitiva no mercado, e a qualidade, nesse ínterim, surgiu como um diferencial necessário.

Agora, somado a isso os fenômenos econômicos, como globalização e abertura de mercados, a qualidade já não é mais percebida como diferencial, e sim como essencial, requisito básico, e uma exigência cada vez maior por parte dos clientes. Dessa forma, o termo “qualidade” foi ganhando muitas definições. Já não bastava buscar e aplicar a qualidade, havia necessidade de comprová-la. Então, como mostrar aos clientes que ainda não conhecem o produto que ele de fato possui qualidade? Ou o que fazer para aqueles que o conhecem não buscarem a concorrência?

Com vários modelos de gestão da qualidade disponíveis, foram sendo criadas empresas para certificar a qualidade, estabelecendo padrões e normas técnicas a serem seguidas. Algumas focadas no produto final, outros no processo de construção do produto, outras ainda focadas nos serviços, na gestão da qualidade, e assim por diante. A linha de normas mais conhecida de certificados é a linha da ISO – *International Organization for Standardization* (Organização Internacional para Padronização).

Conforme já discutimos, definir qualidade é algo muito subjetivo, que depende do momento, do contexto, da situação ou do foco: quando se trata de avaliar uma empresa, produto, serviço ou um profissional, por exemplo.

Analizar a origem da palavra qualidade pode nos dar um ponto de partida sobre o que é qualidade. Originada do latim: *qualitate*, divide-se em duas partes: a base, que seria *qualis*, que pode ser traduzida como “qual?” ou “o quê?”, e *itate*, que significa destaque. Com base nisso, pode-se arriscar dizer que *qualitate* significa “qual o destaque” ou “o que se destaca”.

Contudo, há um consenso: o cliente é quem define a qualidade. A busca pela obtenção da satisfação do cliente é que descreve e direciona a qualidade desejada.

Com a publicação do artigo “O que significa realmente qualidade do produto?” de David A. Garvin, no *MIT Sloan Management Review*, essa visão

é questionada. Para ele, não basta o foco no cliente para definir qualidade, uma vez que é um conceito complexo e multifacetado. Ele propõe uma visão mais ampla e inclusiva, considerando cinco abordagens principais.

1. **Visão transcendental:** qualidade é algo que se reconhece imediatamente, é uma “excelência inata” que só pode ser reconhecida pelo cliente por meio de sua própria experiência com o produto. Porém, ela não pode ser definida explicitamente.
2. **Visão centrada no produto:** a qualidade pode ser medida e definida em relação às características e atributos de um produto, como suas funções e seus recursos.
3. **Visão centrada no valor:** qualidade seria o quanto o cliente está disposto a pagar pelo produto, ou seja, em relação ao nível de conformidade do produto a um custo aceitável.
4. **Visão centrada na fabricação:** qualidade depende da conformidade com os requisitos, conforme estabelecidos pelo projeto do produto.
5. **Visão centrada no cliente ou usuário:** a qualidade é definida segundo as metas ou as necessidades e conveniências do cliente ou usuário. Se o produto atende a essas metas, ele tem qualidade.

Além das cinco visões para definir qualidade, David complementou o tema com oito dimensões da qualidade, com a publicação na *Harvard Business Review*, em 1987, do seu artigo “Competindo nas oito dimensões da qualidade”. Nesse artigo, ele propôs um novo modelo para qualidade, em que cada dimensão é autossuficiente e distinta, agrupando certos atributos de um produto conforme um critério de classificação.

Algumas dimensões poderão ser mais importantes que outras ou poderão ser irrelevantes, dependendo do produto em questão. Mas juntas fornecem uma gama completa de requisitos e expectativas quanto à qualidade do produto. Algumas dimensões são concretas e mensuráveis, enquanto outras são subjetivas, dependendo do cliente. Segundo o artigo de David, as oito dimensões da qualidade, são:

1. **desempenho** – trata do funcionamento do produto;
2. **características** – comprehende aspectos complementares do produto;

3. **conformidade** – grau que as características do produto atendem a padrões especificados;
4. **confiabilidade** – capacidade do produto de deixar de funcionar corretamente;
5. **durabilidade** – compreende a vida útil de um produto.
6. **atendimento** – aspectos que podem afetar a percepção do cliente;
7. **estética** – aparência de um produto;
8. **qualidade percebida** – visão que se tem do fornecedor, a qual se transfere para o produto.

Embora tenha se passado muitos anos, e outros fatores tenham surgido, como a evolução tecnológica, as contribuições de David são utilizadas até hoje (QUALITY WAY, 2015). Com base no contexto histórico da qualidade, é fácil deduzir que na área da computação as coisas não foram diferentes. Especialmente quando se trata de softwares, os quais alcançaram um papel fundamental e crucial na competitividade das empresas e na forma de viver em sociedade.

Hoje se produzem softwares cada vez mais complexos, e a qualidade é um fator essencial no seu desenvolvimento. Da mesma forma que definir qualidade é algo complexo, qualidade de software também enfrenta a mesma questão.

Assim como a indústria se modernizou, nas últimas décadas, a computação se desdobrou em várias subáreas de estudo. A tecnologia evoluiu de tal forma que a quantidade de informação explodiu, exigindo profissionais cada vez mais especializados. Na mesma proporção, a qualidade é cada vez mais valorizada, e as empresas de desenvolvimento de software procuram investir mais nesse quesito do que perder dinheiro com softwares que não funcionam adequadamente.

Na década de 1990, as principais empresas reconheciham que bilhões de dólares por ano eram desperdiçados em software que não apresentava as características e as funcionalidades prometidas. Pior ainda, tanto o governo quanto as empresas estavam cada vez mais preocupados com o fato de que uma falha grave de software poderia inutilizar importantes infraestruturas, aumentando o custo em dezenas de bilhões (PRESSMAN, 2016, p. 412).

Mesmo nos tempos atuais, a qualidade continua a ser um problema, e por mais conhecimento especializado que se tenha, ainda existem fatores que prejudicam o desenvolvimento de um software com qualidade. Segundo os clientes, a culpa da falta de qualidade está nos desenvolvedores devido a práticas descuidadas. Por sua vez, os desenvolvedores culpam os clientes e demais envolvidos, por prazos absurdos, contínuas mudanças, forçando a entrega de um software antes de ele estar completamente validado. Segundo Pressman (2016), ambos estão com a razão: “todos querem construir sistemas de alta qualidade, mas o tempo e o esforço necessários para produzir um software “perfeito” não existem em um mundo orientado ao mercado”.

Não há dúvida de que, assim como para todo produto ou serviço a qualidade é uma característica indispensável, ela também é para o produto gerado pelo desenvolvimento de software.

No anseio de verdadeiramente encontrar uma forma de reduzir as dificuldades e solucionar deficiências no desenvolvimento de um software, como vimos em capítulos anteriores, foi criada a engenharia de software. Conforme Pressman (2016), um dos objetivos da engenharia de software é criar um produto de qualidade, em que uma qualidade melhor leva à redução do retrabalho. E retrabalho reduzido resulta em tempos de entrega menores.

Segundo Rocha (2001), a engenharia de software tem como objetivo desenvolver produtos de alto nível de qualidade. Para isso preocupa-se com a qualidade do produto e a qualidade do desenvolvimento do produto.

Contudo, Rezende (2005) alerta que o software deve satisfazer todos os envolvidos no projeto: clientes, desenvolvedores e usuários. E a insatisfação pode ocorrer nas diferentes etapas do processo de desenvolvimento: análise, projeto, construção, implantação ou manutenção.

Apesar de gerentes e profissionais reconhecerem a necessidade de uma abordagem mais disciplinada para o software, eles continuam a debater a forma pela qual essa disciplina deve ser aplicada. Muitos indivíduos e empresas ainda desenvolvem software ao acaso, mesmo quando constroem sistemas para servir às tecnologias mais avançadas da atualidade. Muitos profissionais e estudantes desconhecem os métodos modernos. Em decorrência disso, a qualidade do software que produzimos é sofrível e coisas ruins acontecem. Além disso, continua o debate e a controvérsia sobre a verdadeira natureza da abordagem de engenharia de software. O estado atual da engenharia de

software é um estudo de contrastes. As atitudes mudaram, houve progresso, mas muito resta a ser feito antes que a disciplina alcance maturidade total (PRESSMAN, 2006, p. 601).

A complexidade para definir a qualidade de software não está apenas nesse ponto, mas também nas formas de se obter e conquistá-la, seja no processo ou no produto. A obtenção da qualidade depende de recursos – requer tempo e dinheiro, e, por vezes, os prazos curtos e orçamentos enxutos fazem com que sejam puladas etapas dentro dos processos de desenvolvimento de software. Se instala, com isso, o chamado dilema da qualidade: se é produzido um software de péssima qualidade de modo a atender ao prazo e ao orçamento, os impactos podem ser muito negativos, com a possibilidade de ficar com um produto encalhado; mas se para obter um software de alta qualidade é necessário aplicar grandes esforços, com altas somas de dinheiro e ficar postergando o prazo de entrega, também pode-se ter impactos negativos, como a falta de credibilidade por parte dos clientes, chegando à falência do projeto ou da empresa (PRESSMAN, 2016).

A busca e manutenção da qualidade não é uma tarefa trivial, ela é exigente e árdua, requer disciplina, foco e validações constantes. Logicamente, sempre irá se deparar com desafios. Um dos primeiros desafios a serem enfrentados, por estarem exigindo sempre atenção e cuidado, é a definição da qualidade, pois não é um conceito ou ideia que permanece imutável ao longo dos anos, deve sempre ser adequada ao contexto do ambiente e do mercado. Especialmente no que diz respeito ao produto de software, pois ele tem características difíceis de serem especificadas, mas que são de grande relevância, por exemplo: a garantia de segurança dos dados, como determinar e documentar sobre eficiência, como escrever requisitos sobre a facilidade de manutenção e facilidade de uso.

Além do fato de que o desenvolvimento de software é ainda realizado de modo manual e tem aspecto não repetitivo, com requisitos voláteis, que constantemente estão sendo alterados, e nem sempre se repetem. Depender muito dos requisitos faz com que facilmente sejam apresentadas falhas no levantamento de requisitos, uma vez que o próprio cliente tem dificuldades para expressá-los, ou esquece-se de mencionar, ou simplesmente pensa que deve ser óbvio para os profissionais que desenvolvem o software, ou seja, há muitos requisitos implícitos.

Afinal, um software não apresenta uma forma única de construção. É um produto que depende muito do ser humano, e que constantemente exige tomada de decisão, fazendo-se escolhas entre o desejável, o possível e o conhecido, podendo apresentar a necessidade de se alterar, excluir ou incluir novos requisitos em etapas já avançadas do desenvolvimento, o que torna o desenvolvimento do software em si, algo muito complexo.

Para facilitar essa questão, alguns autores criaram critérios que auxiliam no momento de definir características de qualidade desejadas para o produto de software, conhecidas como dimensões e fatores de qualidade de software.

10.2 Dimensões e fatores de qualidade

Como visto anteriormente, as dimensões de qualidade foram desenvolvidas por David Garvin, que sugeriu que “a qualidade deve ser considerada adotando-se um ponto de vista multidimensional que começa com uma avaliação da conformidade e termina com uma visão transcendental (estética)” (PRESSMAN, 2016). Tais dimensões não foram criadas especificamente para o produto de software, mas Pressman (2016) as adequou para qualidade de software, da forma a seguir.

1. **Qualidade do desempenho** – o software fornece todo o conteúdo, funções e recursos especificados como parte do modelo de requisitos, de modo a gerar valor ao usuário?
2. **Qualidade dos recursos** – o software fornece recursos que surpreendem e encantam usuários que os utilizam pela primeira vez?
3. **Confiabilidade** – o software fornece todos os recursos e capacidades sem falhas? Está disponível quando necessário? Fornece funcionalidade sem a ocorrência de erros?
4. **Conformidade** – o software está de acordo com os padrões de softwares locais e externos relacionados com a aplicação? Segue as convenções de projeto e codificação de fato?

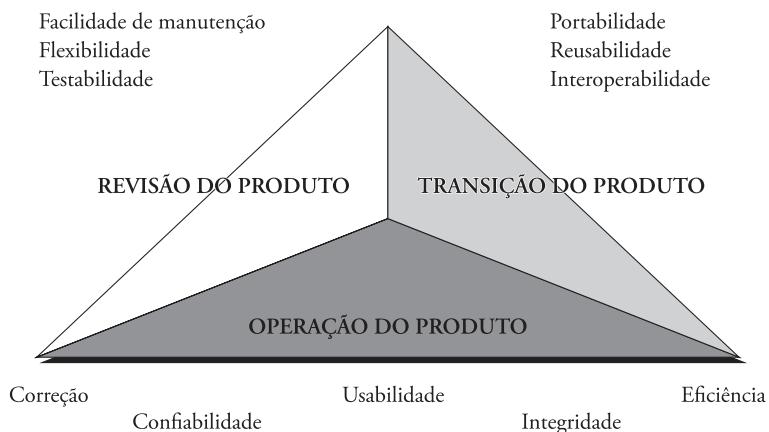
5. **Durabilidade** – o software pode ser mantido (modificado) ou corrigido (depurado) sem a geração involuntária de efeitos colaterais indesejados? As mudanças farão com que a taxa de erros ou a confiabilidade degradem com o passar do tempo?
6. **Facilidade de manutenção** – o software pode ser mantido (modificado) ou corrigido (depurado) em um período de tempo aceitável e curto? O pessoal de suporte pode obter todas as informações necessárias para realizar alterações ou corrigir defeitos?
7. **Estética** – a maioria de nós concordaria que uma entidade estética tem certa elegância, um fluir único e uma “presença” que são difíceis de quantificar, mas que, não obstante são evidentes.
8. **Percepção** – a nossa percepção de qualidade pode ser influenciada por preconceitos ou e ideias pré-definidas.

Porém, Pressman (2016) comprehende que muitas dessas dimensões podem ser consideradas apenas subjetivamente, e por essa razão, é importante utilizarmos um conjunto de fatores de qualidade que podem ser classificados em duas grandes categorias: (1) fatores que podem ser medidos diretamente (por exemplo, defeitos relevantes durante os testes) e (2) fatores que podem ser medidos apenas indiretamente (por exemplo, usabilidade ou facilidade de manutenção).

Para definir os atributos de qualidade que um software deve ter de modo a atender as necessidades de seus usuários, alguns “fatores de qualidade de software” foram definidos e podem variar em quantidade e definição, conforme o autor ou a organização. Os fatores de qualidade de software devem, de preferência, ser possíveis de mensurar podendo constituir um ou mais dos requisitos de um software (RIOS, 2013).

Em 1979, McCall, Richards e Walters criaram uma proposta de categorização para os fatores de qualidade de software, como podemos observar na figura 10.1.

Figura 10.1 – Fatores da qualidade de McCall



Fonte: Pressman (2016).

A seguir, podemos observar no quadro 10.1 o detalhamento desses fatores com os respectivos indicadores.

Quadro 10.1 – Fatores da qualidade de McCall

Fator	Conceito	Fórmulas de Cálculo do Indicador
Correção	Capacidade do programa de satisfazer suas especificações e atender aos objetivos dos usuários	1 (erros/linhas de código)
Eficiência	Quantidade de recursos computacionais e código requerido por um programa para apoiar uma função	1 (atual utilização/utilização alocada)
Flexibilidade	Esforço requerido para modificar um programa em operação	1 – 0.05 (média de dias de trabalho para mudar)
Integridade	Capacidade de controlar o acesso aos dados por pessoa não autorizada.	1 (falhas/linha de código)
Interoperabilidade	Esforço requerido para integrar um sistema com outro	Falhas – relativas à segurança

Fator	Conceito	Fórmulas de Cálculo do Indicador
Manutenibilidade (ou facilidade de manutenção)	Esforço requerido para localizar e corrigir um defeito em um programa em produção	1(esforço para integrar/esforço para desenvolver)
Portabilidade	Esforço requerido para transferir um programa de uma configuração de hardware e/ou ambiente operacional de software para outro	1 – 0.1(média de dias de trabalho para localizar e corrigir)
Confiabilidade	Capacidade de atender à expectativa de que um programa execute suas funções com a precisão requerida	1 (esforço para portar/esforço para desenvolver)
Reuso	Capacidade de que um programa (ou componente) possa ser usado em outra aplicação	1 (defeitos/linhas de código)
Testabilidade	Esforço requerido para testar um programa para garantir que ele executa suas funções	1 (esforço para converter/esforço para desenvolver)
Usabilidade	Esforço requerido para alguém usar, ou seja, aprender, preparar a entrada de dados e interpretar os resultados de um programa	1 (esforço para testar/esforço para desenvolver)

Fonte: Rios (2013).

Segundo Pressman (2016), o padrão ISO 9126 representa a atual padronização mundial para a qualidade de produtos de software, e foi desenvolvido para identificar os atributos fundamentais de qualidade para o software de computador.

- Funcionalidade:** o grau com que o software satisfaz as necessidades declaradas, conforme indicado pelos atributos – adequabilidade, precisão, interoperabilidade, conformidade e segurança.
- Confiabilidade:** período de tempo que o software fica disponível para uso, conforme indicado pelos seguintes subatributos – matu-ridade, tolerância a falhas e facilidade de recuperação.

3. **Usabilidade:** o grau de facilidade de uso do software, conforme facilidade de compreensão, facilidade de aprendizado e operabilidade.
4. **Eficiência:** grau de uso otimizado dos recursos por parte do software, dos recursos do sistema, conforme comportamento em relação ao tempo, comportamento em relação aos recursos.
5. **Facilidade de manutenção:** a facilidade com a qual podem ser feitas correções no software, conforme facilidade de análise, facilidade de realizar mudanças, estabilidade e testabilidade.
6. **Portabilidade:** facilidade com a qual o software pode ser transportado de um ambiente para outro, conforme adaptabilidade, facilidade de instalação, conformidade e facilidade de substituição.

No entanto, não é tão simples determinar quando um software está com qualidade. Em face dessa dificuldade, Pressman (2016) diz que não basta entender que a qualidade é importante, é preciso:

- × definir explicitamente o que realmente se quer dizer com qualidade de software;
- × criar um conjunto de atividades que ajudarão a garantir que cada componente desenvolvido exiba alta qualidade;
- × utilizar métricas visando desenvolver estratégias para melhoria do processo de software e, consequentemente, do produto final.

Com base nessa visão e na busca de um olhar mais sistêmico, comprehende-se que o ideal é que a busca pela qualidade seja constante e que ela não deve se concentrar somente no início do projeto de desenvolvimento de software ou em outra fase. Bartié (2002) já compreendia isso, e definiu que «qualidade não é uma fase do ciclo de desenvolvimento de software... é parte de todas as fases». Portanto, a qualidade deve permear todas as fases, bem como ela deve ser foco também de todos os envolvidos no desenvolvimento do software (PRESSMAN, 2016).

Foi com essa visão, que muitos conceitos foram sendo desenvolvidos a respeito do que seria a qualidade de software, entendendo que não basta focar no produto, é preciso olhar também o processo de desenvolvimento dele. A seguir são destacados alguns deles.

- ✖ De acordo com o SWEBOK (*Guide to the Software Engineering Body of Knowledge*) (2011), documento criado pelo IEEE, para servir de referência em assuntos considerados pertinentes à engenharia de software, “a qualidade de um produto está na qualidade do processo que foi utilizado para criá-lo”.
- ✖ Para Philip Crosby (1992), qualidade é a conformidade com requisitos e especificações, a qual é medida pelo custo da não conformidade. Uma ideia fortemente ligada ao conceito de defeito zero e os custos da falta de qualidade.
- ✖ Já para Bartié (2002), “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”.

Essas definições deixam claro que a qualidade de software deve ter duas frentes de trabalho bem distintas: qualidade do processo e qualidade do produto, e que uma depende fortemente da outra. Portanto, é primordial que a organização não foque apenas em uma frente ou em outra, já que a qualidade não é algo que pode ser inserido em um software depois de ele estar pronto, e que, para que ele a tenha, não há outra forma senão introduzir a qualidade no produto de software a partir do seu processo de desenvolvimento.

Afinal, ter um processo de qualidade, por si só não garante que os produtos desenvolvidos possuem qualidade, mas já sugerem que a organização é capaz de produzir bons softwares, e não é possível se obter qualidade espontaneamente. Ter um processo de desenvolvimento de software de qualidade permite à empresa aumentar a qualidade de seus produtos, reduzir retrabalhos, ter maior produtividade, reduzir o tempo para atender o mercado, aumentar a sua competitividade e obter maior precisão nas estimativas.

Ou você faz certo da primeira vez ou faz tudo de novo. Se uma equipe de software buscar a qualidade em todas as atividades de engenharia de software, a quantidade de retrabalho será reduzida. Isso resulta em custos menores e, mais importante, menor tempo para disponibilização do produto no mercado (PRESSMAN, 2016, p. 412).

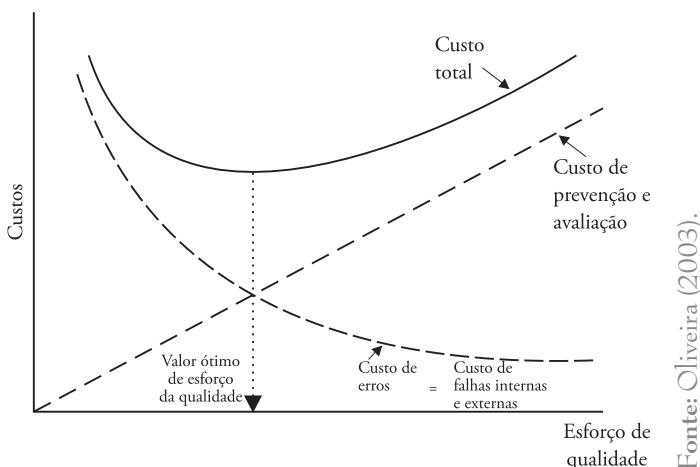
Contudo, como já foi visto, não é suficiente que a empresa defina as características e aspectos relevantes para a qualidade do seu produto e de seu

processo de desenvolvimento. Ela precisa também encontrar formas de garantir que essa qualidade seja percebida e que ela não fique ultrapassada, ou não esteja contemplando as exigências do mercado de software. Para conseguir obter isso, é necessário que a empresa desenvolva dentro da sua organização um método para gerenciar, garantir e controlar essa qualidade.

Para tanto, é necessário que exista planejamento e investimento adequados para garantir que a qualidade de software desejada seja obtida, conforme os padrões de qualidade estabelecidos. A qualidade tem um custo e, se a empresa relutar em investir em um programa de qualidade, ela deve analisar quais serão os efeitos da falta de qualidade e refletir que essa falta de qualidade também tem um custo.

[...] sabemos que a qualidade é importante, mas ela nos custa tempo e dinheiro –tempo e dinheiro demais para obter o nível de qualidade de software que realmente desejamos. Independentemente da estratégia escolhida, a qualidade tem, efetivamente, um custo que pode ser discutido em termos de prevenção, avaliação e falha. Os custos de prevenção. [...] Não há dúvida nenhuma de que a qualidade tem um preço, mas a falta de qualidade também tem um preço – não apenas para os usuários, que terão de conviver com um software com erros, mas também para a organização de software que o criou e, além de tudo, terá de fazer a manutenção. A verdadeira questão é a seguinte; com qual custo devemos nos preocupar? (PRESMAN, 2016, p. 422).

Figura 10.2 – Relação entre os custos da qualidade



Fonte: Oliveira (2003).

Não há dúvida de que a qualidade é importante e deve ser priorizada nos projetos de desenvolvimento de software. Porém, a empresa deve considerar buscar apoio caso não tenha profissionais com conhecimento técnico sobre o assunto dentro de sua organização. Ela não deve sair às cegas sem um planejamento e com um orçamento adequado à sua capacidade financeira e criar um programa de qualidade sem o devido conhecimento, pois poderá ter custos financeiros indesejados e não alcançar seu objetivo. A qualidade não é algo que possa ser obtido de qualquer maneira. É preciso encará-la com seriedade e buscar embasamento. Desenvolver um programa de qualidade começa por definição e implantação de um processo de software. E um processo de software deve ser documentado, compreendido e seguido.

Atualmente existem diversos modelos e padrões de qualidade que englobam todas as fases do ciclo de vida do software visando garantir a qualidade do processo como a qualidade do produto. Investir em qualidade e num sistema de gestão da qualidade é primordial para uma empresa se tornar competitiva no setor de desenvolvimento de software, especialmente se alcançar padrões de qualidade reconhecidos em território nacional e internacional. Além disso, as empresas que priorizam a qualidade, tanto nos seus processos quanto no seu produto, conseguem oferecer um preço mais competitivo e obter confiança do seu cliente e de seus fornecedores. Mas para isso, não basta que a qualidade exista, ela deve ser percebida e reconhecida pelo cliente. Uma das maneiras de se fazer isso seria por meio de certificações oficiais de padrão de qualidade.

Para isso, existem empresas responsáveis por reconhecer a qualidade de acordo com alguma norma ou padrão e que estão autorizadas pelos órgãos responsáveis pela definição da norma ou padrão técnico de produção. No setor de desenvolvimento de software, as principais referências para melhoria de processos utilizada são o modelo Capability Maturity Model (CMM) e as normas ISO 9000, ISO 12207 e ISO 15504.

Cada empresa deve definir qual o melhor modelo de gestão de qualidade seguir, com base em suas estratégias de negócio e de mercado, e considerando os benefícios e o retorno sobre investimentos realizados. Cada modelo tem suas características peculiares, pois são idealizados com base em diversas considerações e práticas, e por isso cada um apresenta vantagens e desvantagens

no momento da implantação e da implementação, o que também deve ser considerado na decisão da escolha do mais apropriado a cada organização (BANAS QUALIDADE, 2004).

10.3 Planejamento e gerência da qualidade de software

Segundo o PMBOK (*Project Management Body of Knowledge*), o gerenciamento da qualidade do projeto deve considerar todos os processos necessários de modo a assegurar que ele atenda plenamente às necessidades para as quais foi criado.

No caso do desenvolvimento de software, a qualidade dificilmente pode ser incorporada após o produto finalizado, e, do levantamento dos requisitos do usuário à entrega do produto final, existe um processo de desenvolvimento que frequentemente envolve uma série de etapas, as quais, se não gerenciadas, podem comprometer a qualidade do produto final. Sendo assim, a qualidade do software deveria ser o objetivo do processo de desenvolvimento, e desse modo, ter previamente estabelecidas as características de qualidade que se deseja alcançar durante o desenvolvimento permitirá produzir um produto de software com qualidade.

Entender a qualidade do processo de desenvolvimento de software está diretamente ligado à engenharia de software. Estudar modelos de processo de desenvolvimento de software auxilia a compreender detalhes sobre como se desenvolve um software e quais são as etapas envolvidas. Sem a compreensão de cada pequena tarefa envolvida no desenvolvimento, torna-se difícil ou até mesmo equivocada a escolha do melhor modelo de gestão de qualidade. Os principais processos para o gerenciamento da qualidade, com base ainda no PMBOK, estão descritos a seguir.

1. Planejamento da qualidade (identificação): contém objetivos mensuráveis, critérios de entradas e saídas, identifica quais os padrões de qualidade relevantes para o projeto, critérios de verificação e validação, responsabilidades, configuração controles de alteração, correção e defeitos. Fomentam o planejamento informações fundamentais como: política da qualidade da organização, definição do

escopo, processos, modelos e técnicas definidas no âmbito organizacional para o desenvolvimento do software, padrões regulamentares fora do escopo da organização e descrição do produto.

2. Nesse escopo devem ser considerados os padrões de qualidade legais, definidos por normas e legislação, além dos padrões exigidos pelo cliente. Dessa forma, o processo de planejamento explicita as ações dos processos de garantia e de controle de qualidade e deve incluir a documentação para as atividades de garantia e controle, como inspeções, testes, coleta de métricas, além do uso de padrões e descrição do processo de software adotado. E ainda deve desenhar o plano da qualidade de software, no qual fornecerá informações sobre como a equipe de qualidade irá garantir o cumprimento dos padrões de qualidade estabelecidos para o produto de software. O plano da qualidade pode ser um documento específico ou parte do plano do projeto de desenvolvimento e deve ser utilizado como base do gerenciamento ao longo de todo o ciclo de vida do projeto de desenvolvimento do software.
3. Garantia da qualidade ou Software Quality Assurance (SQA) (avaliação do resultado): tem por objetivo avaliar constantemente o desempenho geral do projeto, de modo a garantir o sucesso do projeto em alcançar os padrões definidos de qualidade. Consiste de diversas atividades de apoio planejadas em todo o processo de desenvolvimento do software com a função de garantir o desempenho de cada uma das fases do desenvolvimento, conforme os padrões de qualidade previamente definidos. Os processos de desenvolvimento devem estar definidos de maneira clara e documentados, e é primordial que as pessoas os sigam rigorosamente (STAA, 2000). É relevante que a gerência esteja comprometida com as técnicas de garantia de qualidade e exija que todos os membros envolvidos no processo também estejam. Essa fase visa encontrar defeitos antes de introduzi-los no software, ou seja, a prevenção. Deve-se, portanto, aplicar métodos e medidas técnicas sólidas, conduzir revisões técnicas bem definidas e testes bem elaborados, a fim de evitar a introdução de defeitos durante o desenvolvimento do software e, com isso, aumentar a probabilidade de o software apresentar uma

qualidade satisfatória antes mesmo do primeiro controle de qualidade. “Aqui estarão relacionados os testes de verificação (testes estáticos) e os testes de validação (testes dinâmicos ou testes de softwares) previstos em todas as etapas do ciclo de desenvolvimento” (BARTIÉ, 2002). Para que a garantia da qualidade seja efetivada, além de atuar sobre o processo de desenvolvimento, precisa revisar e auditar a qualidade dos artefatos produzidos e dos processos de desenvolvimento para verificar se estão conforme padrões e requisitos definidos. É nesse contexto que entra o controle de qualidade.

4. Controle da qualidade (controlar os resultados): monitorar os resultados obtidos do projeto, avaliando se estão de acordo com os padrões definidos de qualidade, e identificar formas de eliminar ou contornar as causas de resultados negativos. Com o controle, é possível evitar que softwares defeituosos sejam entregues aos clientes e monitorar o processo identificando e corrigindo defeitos, por meio de atividades como: inspeções, simulações e testes. Essa fase, portanto, visa encontrar defeitos que já foram introduzidos no software, ou seja, detectar e corrigir os defeitos. O controle de qualidade aplica-se durante o processo de desenvolvimento, podendo ser de responsabilidade dos próprios desenvolvedores, ou, se possível, de uma equipe independente da empresa – esta muito dependente do plano de garantia da qualidade – e incluir atividades de verificação, validação e aprovação. Por isso, é importante, que os artefatos produzidos estejam com especificações de requisitos bem definidas e mensuráveis, permitindo fazer uma comparação do resultado de cada um (PRESSMAN, 2006). Segundo Lobo (2009), “O controle de qualidade de software é um dos fatores mais importantes em uma fábrica de software. Manter um software com qualidade é garantia de novos clientes e satisfação dos atuais”.

10.4 Modelos e normas de gestão da qualidade de software

Para auxiliar na melhoria de processos de desenvolvimento e promover a normatização de produto e serviço, foram criados modelos e normas de

qualidade de software com o propósito de atender plenamente aos requisitos de qualidade. Por meio deles, a empresa determina os padrões de qualidade que irá seguir.

Os padrões e as normativas permitem medir diversos aspectos da qualidade de software, como: qualidade de produto de software, qualidade do processo de desenvolvimento e nível de maturidade da organização.

De início, para implantar um programa de qualidade, é necessário a definição e a documentação do processo utilizado para o desenvolvimento do software com as atividades a serem realizadas, a estrutura e a organização, os artefatos necessários e os que serão produzidos, bem como os recursos necessários (humanos, hardware e software) para a realização das atividades.

Um padrão é efetivo se, quando usado apropriadamente, melhora a qualidade dos resultados e reduz os custos dos produtos de software (FENTON, 1996).

A seguir, vamos conhecer algumas normas aplicadas para a qualidade de software.

10.4.1 ISO 9000 e ISO 9000-3

As normas da família NBR ISO 9000 referem-se a um padrão internacional de qualidade desenvolvida para ajudar as organizações, independentemente do ramo e tamanho, na implementação e na operação da gestão da qualidade, de modo a gerar produtos que atendam às expectativas de seus usuários. Elas seguem a premissa de que se o processo de produção e a gestão são de boa qualidade, então o produto/serviço também será de boa qualidade.

A sigla ISO significa Organização Internacional para Normalização (*International Organization for Standardization*) e a organização principal está localizada em Genebra, na Suíça. De acordo com Schmauch (1994), quando um sistema de qualidade seguir os padrões da ISO 9000, garantirá que seu processo de desenvolvimento disponha de um nível de controle, disciplina e padrão, garantindo a qualidade de seus produtos.

Para os processos de software, foi criada o guia ISO 9000-3, com diretrizes para aplicar a ISO 9000 adequadamente ao software, ou seja, especificamente para a área de desenvolvimento, fornecimento e manutenção

de software. As diretrizes propostas visam atender a questões que giram em torno de uma “situação contratual”, na qual uma empresa contrata outra para desenvolver um produto de software.

- × **Estrutura do sistema de qualidade:** quais as responsabilidades do fornecedor e do comprador com análise crítica conjunta.
- × **Atividades do ciclo de vida:** análise crítica do contrato, especificação dos requisitos do comprador, projeto e implementação com a definição de metodologias, métodos de verificação e validação, critérios de aceitação, versionamento, quantidade de originais e cópias, procedimentos para entrega, instalação e manutenção.
- × **Gerenciamento do projeto:** gestão do início ao fim do projeto, gerenciamento de configuração, registros e documentação, medição, convenções, finalização, entregas de produto e treinamento.

Essa norma tem uma limitação no que diz respeito à melhoria contínua do processo de software praticada por outros modelos como o CMM. Ela trabalha muito com quais processos a organização deve ter e manter, mas deixa a desejar, pois não indica os passos necessários para desenvolver tais processos e nem como melhorá-los.

Também há outras séries da ISO voltadas para o processo de desenvolvimento de software, como a ISO/IEC 12207 e a ISO/IEC 15504 PDTR.

A Norma ISO/IEC 12207 está voltada para os processos de ciclo de vida de software; foi criada em 1995 e atualizada em outubro de 2001 com algumas melhorias. Ela ajuda a estabelecer uma estrutura para os processos de ciclo de vida e de desenvolvimento de software. A norma cobre todo o ciclo de vida do desenvolvimento de software, desde os requisitos até a manutenção, oferecendo melhoria dos processos com foco em qualidade, orçamentos, prazos e recursos definidos no projeto. A ISO/IEC 12207 determina três categorias distintas para os processos de desenvolvimento: primários, de apoio e organizacionais. A norma descreve cada um desses processos, que são compostos por um conjunto de atividades, e cada atividade é composta por um conjunto de tarefas. Ela pode ser adaptada para qualquer empresa.

Já a norma ISO/IEC 15504 se preocupa com a avaliação dos processos de software. Surgiu justamente de um estudo que buscava desenvolver um

padrão voltado para sanar a necessidade de avaliação de processos de software, com foco em criar a norma e, com isso, suprir demandas e requisitos de um padrão internacional de avaliação dos processos de software. Para Salviano (2003), a norma ISO/IEC 15504 trata-se de um framework para avaliação de processos de software, organizando e classificando as melhores práticas em duas dimensões: categorias de processo e níveis de capacidade. Porém, atualmente a norma se tornou genérica e pode ser aplicada por diversos tipos de processos, não sendo mais exclusiva para o software.

10.4.2 CMM e CMM-I

O CMM (Capability Maturity Model) é um modelo que visa medir a maturidade das empresas que desenvolvem software, avaliando a capacidade da empresa de desenvolver software. Surgiu como guia para ajudar as organizações de software a melhorarem seu processo, visando aumentar a capacitação ou capacidade de seu processo de desenvolvimento de software (PESSOA, 2003).

O CMM surgiu nos Estados Unidos, como iniciativa do SEI (Software Engineering Institute), com o objetivo de avaliar e melhorar a capacitação de empresas que desenvolvem software, tendo o apoio do Departamento de Defesa do Governo dos Estados Unidos, que, como um grande consumidor de produtos de software, necessitava de um modelo formal para fazer a seleção de seus fornecedores.

Embora emitido por instituição internacional, o modelo tem sido bem aceito até mesmo fora do mercado americano. Ele foi publicado em 1992 e está disponível na internet. Foca na documentação dos processos, buscando a melhoria por meio da adaptação destes à empresa ou aos projetos desenvolvidos pela empresa, evitando desorganização e inexistência de padrões documentados. Esse modelo, no entanto, não busca resolver problemas encontrados na organização, mas se propõe a ajudar as organizações a encontrarem suas próprias soluções. Descreve cinco estágios de maturidade das organizações, enquanto evoluem seu ciclo de desenvolvimento de software:

1. **inicial** – nessa fase pode existir um processo de desenvolvimento, mas ele é totalmente ignorado, com pulo de etapas e uma qualidade suspeita;

2. **repetível** – tem um processo definido, mas que nem sempre é seguido. E na maioria das vezes em que é seguido, é devido a momentos de estresse;
3. **definido** – tem processo bem definido e compreendido, documentado, com aplicação de padrões, procedimentos, ferramentas e métodos, mas não é gerenciado;
4. **gerenciado** – além do processo bem definido, com suporte de ferramentas e métodos, também apresenta métricas precisas para avaliar e controlar o processo conforme foi idealizado, com possibilidades de ajustes a novos projetos;
5. **otimizado** – apresenta um processo maduro, com gerenciamento e também tem avaliações constantes buscando melhorias contínuas no desempenho dos processos.

Já o CMMI (Capability Maturity Model Integration) também surgiu de um estudo desenvolvido pelo SEI, que buscava compatibilizar o modelo CMM com a norma ISSO 15504. O CMMI é um modelo que visa definir e melhorar a capacidade e maturidade dos processos de desenvolvimento do software, fornecendo modelos de melhores práticas que beneficiam a eficácia, a eficiência e a qualidade da organização, com a identificação de pontos fortes e fracos da organização de processos e permitindo mudanças para transformar os pontos fracos em forças (SEI, 2011). Esse modelo auxilia na redução de defeitos e na maior qualificação do pessoal, de modo que as empresas possam reduzir retrabalhos, custos e tempo de entrega dos produtos. Também avalia a empresa com base em cinco níveis de maturidade, similares ao CMM.

10.4.3 MPS.BR

Conhecido como Modelo MPS.BR (melhoria de processo do software brasileiro), busca atender a empresas de software brasileiras de médio a micro-porpe, com custo de certificação reduzido, procurando suprir suas necessidades e promovendo reconhecimento nacional e internacional como modelo de desenvolvimento de software. Foi construída com base no CMMI e nas normas ISO 12207 para desenvolvimento de software, e ISO 15504 para avaliação de processos de software, e é o modelo que está mais adequado à realidade do mercado brasileiro. Apresenta como diferencial a sua escala de implementação,

que se divide em sete níveis de maturidade, permitindo que as empresas se certifiquem em estágios graduais, chegando a um nível inicial de maturidade e capacidade, com um grau menor de esforço e de investimento.

10.5 Melhores práticas da engenharia de software

O uso de boas práticas de engenharia de software contribui para o aumento da qualidade do produto de software e possibilita garantir que o software também agregue valor e atenda às necessidades do negócio. Por isso, fazem parte das boas práticas da engenharia de software e que contribuem significativamente para a qualidade do produto de software:

- ✖ envolver os clientes “donos do negócio” não apenas em etapas de levantamento de requisitos, mas também nas etapas de desenvolvimento, dando feedbacks e tirando dúvidas, ajudando na construção de cenários para testes e testes de aceitação;
- ✖ realizar revisões no código constantemente por meio de programação em pares. As revisões permitem ter códigos mais enxutos e que podem ser reaproveitados ou apresentam uma fácil manutenção;
- ✖ aplicar padrões de codificação;
- ✖ aplicar técnicas de automatizados, como desenvolvimento orientado a testes (TDD);
- ✖ fazer um acompanhamento constante do cliente e sua visão do produto de software;
- ✖ sempre que possível, procurar desenvolver um software que possa ser reutilizado e que possa ser portado para uma gama de equipamentos diferentes.

Síntese

Nesse capítulo, tivemos uma visão abrangente sobre qualidade, um contexto histórico sobre a evolução da computação e, com isso, uma breve visão sobre a origem da engenharia de software, enfocando a qualidade de software

e sua complexa definição. Sem um sistema de gestão da qualidade, a qualidade tão buscada pelas organizações tende a não se tornar uma realidade. Desenvolvimento de software, sendo uma atividade muito dependente do ser humano, requer uma avaliação contínua com foco em qualidade, com métodos e técnicas para garantir e controlar a qualidade desejada. Qualidade de software não deve se voltar apenas para o produto de software e também para o processo de desenvolvimento de software. Vimos alguns modelos de gestão de qualidade aplicados dentro e fora do país, como ISO 9000-3, CMM, CMMI e MPS.BR. Foram citados alguns exemplos de boas práticas para desenvolvimento de software com qualidade.

Atividades

1. Julgue as afirmações a seguir como verdadeiras ou falsas.
 - a) () O principal objetivo da engenharia de software é a qualidade do software.
 - b) () Para obter a qualidade do produto de software, basta a empresa aplicar testes de validação.
 - c) () Modelos de qualidade não se aplicam a processos de software.
 - d) () A qualidade do software é algo de fácil definição e obtenção.
 - e) () O software não precisa de critérios de qualidade por ser um produto que atende facilmente os clientes.
2. Explique por que o processo de desenvolvimento de software necessita de garantia e controle de qualidade.
3. A qualidade não faz falta, pois os custos para implantar uma gestão de qualidade são altos, e os custos da falta da qualidade não apresentam impactos para o produto de software. Essa afirmação está correta? Justifique sua resposta.
4. Testes de software é a única prática indicada para garantir a qualidade de software? Justifique sua resposta.

Conclusão

Nesta obra, procurou-se demonstrar a você a importância da disciplina de Engenharia de Software e como ela pode ser aplicada durante o desenvolvimento de um sistema. Pensando em facilitar seu entendimento, foram utilizados exemplos e um linguajar que fazem parte do nosso cotidiano na área de tecnologia da informação.

Dessa maneira, vimos que a engenharia de software surgiu da necessidade de se construir software com mais qualidade em menor tempo, sendo esta compreendida como uma disciplina de engenharia relacionada a todos os aspectos da produção de software.

Entendemos que os requisitos devem estabelecer o que o software deve fazer e definir também as restrições do seu funcionamento e implementação.

Discutimos que o projeto de software é a fase de desenvolvimento, na qual são feitos modelos com todas as entidades que serão construídas posteriormente a partir dos requisitos do sistema. O projeto de software trabalha com quatro níveis de detalhamento: dados, arquitetura, interface e componentes.

Constatamos que, na fase de construção de software, o engenheiro de software deve procurar minimizar a complexidade, buscar antecipar a mudança, pensar em construir para a verificação, utilizar o reuso e aplicar padrões em construção.

Compreendemos que, devido às constantes mudanças que permeiam o ciclo de vida do desenvolvimento de software, é necessária uma forma de organização e controle de todos os elementos envolvidos nessas mudanças e, para isso, existem inúmeras soluções que incluem propostas de processos, métodos e ferramentas.

Por fim, entendemos que a qualidade de software não deve se voltar apenas para o produto de software, mas também para o processo de desenvolvimento de software, sendo os testes de software fundamentais para garantir a qualidade do produto.

“A sociedade, cada vez mais, depende de sistemas de software avançados, portanto é preciso que os engenheiros de software sejam capazes de produzir sistemas confiáveis de maneira econômica e rápida.”

(SOMMERVILLE, 2011, p.5)

Gabarito

1. Introdução à Engenharia de Software

1. A Engenharia de Software é executada por engenheiros de software e sua utilização possibilita que o desenvolvimento de software seja executado dentro do prazo e com alta qualidade.
2. Um processo de software pode ser definido como um conjunto de ferramentas para o desenvolvimento de um produto de software; por intermédio dessas ferramentas é possível planejar e controlar a gestão do projeto de software.
3. Os modelos sequenciais propõem a entrega completa do sistema, ou seja, a entrega é realizada somente após a conclusão de todas as atividades do desenvolvimento. Já nos modelos incrementais, a liberação é realizada em uma série de versões, denominadas incrementos, que oferecem progressivamente maior funcionalidade ao cliente à medida que cada incremento é entregue.
4. O código de ética e prática profissional da Engenharia de Software pode servir de inspiração para o ideal da profissão, normalizando os procedimentos e facilitando a tomada de decisões, além de elevar a imagem da profissão junto à comunidade.

2. Requisitos de Software

1. b, d, a, c, e
2. a = Requisito funcional; b = Requisito funcional; c = Requisito funcional; d = Requisito não funcional; e = Requisito não funcional.
3. Requisitos de produto; requisitos organizacionais e requisitos externos.
4. Clientes, usuários e especialistas de domínio; gerentes de cliente; gerentes de fornecedor; desenvolvedores e testadores.

3. Projeto de Software

1. O projeto de software aproxima o sistema e o mundo real, auxiliando a garantir a qualidade do produto.

2. Projeto de dados: tem por objetivo projetar a estrutura de armazenamento de dados necessária para implementar o software. Projeto arquitetural: visa definir os grandes componentes estruturais do software e seus relacionamentos. Projeto da interface: descreve como o software deverá se comunicar dentro dele mesmo, com outros sistemas e com pessoas que o utilizam (interface com o usuário). Projeto como componente: tem por objetivo refinar e detalhar a descrição dos componentes estruturais da arquitetura do software.
3. Confiabilidade diz respeito à capacidade de o software manter seu nível de desempenho, sob condições estabelecidas, por um período de tempo.
4. Coesão é uma indicação do grau em que um módulo focaliza apenas uma coisa.

4. Construção de Software

1. Reuso significa o uso de elementos (bibliotecas, módulos, componentes e código-fonte) existentes em problemas diferentes. “Construção para reutilização” significa criar elementos de software reutilizáveis, enquanto “construção com reutilização” significa reutilizar elementos de software na construção de uma nova solução ou um novo projeto.
2. O uso de padrões ajuda a alcançar os objetivos do projeto em termos de eficiência, qualidade e custo.
3. O objetivo dos testes de construção é diminuir o intervalo entre o momento em que as falhas são inseridas no código e o momento em que essas falhas são detectadas, reduzindo assim o custo da correção. Os tipos mais comuns são teste unitário e teste de integração.
4. A documentação do código-fonte é um item essencial tanto para atividades de validação do software quanto para as tarefas de manutenção.

5. Testes de Software

1.
 - a) (F) O principal objetivo dos testes é revelar defeitos não conhecidos, e não provar a ausência deles.
 - b) (V) O teste de sistema visa validar os requisitos especificados, procurando simular o ambiente do usuário final de modo que, para isso, o software já deve estar com a estrutura final que será utilizada pelos usuários.
 - c) (F) Os testes funcionais podem ser automatizados, ao contrário dos testes estruturais, que apresentam mais complicações para isso.
 - d) (F) Os testes têm dois enfoques, sendo um deles o de verificação, que pode ser iniciado assim que as especificações forem sendo construídas.
 - e) (F) Testes não são atividades triviais e para uma efetividade devem ser muito bem planejados, estruturados e com técnicas adequadas para cada uma das fases existentes: testes de unidade, testes de integração, testes de sistema e testes de aceitação e realizado por profissionais qualificados.
2. O teste de aceitação é o último nível de testes e pode ser realizado por um grupo restrito de usuários finais ou não.
3. A partir do teste unitário, no qual já devem existir funções e/ou trechos já codificados.
4. O teste funcional, cuja preocupação é validar o funcionamento e o comportamento do software em relação aos requisitos.

6. Manutenção de Software

1. Para continuarem atendendo às necessidades do negócio, os sistemas devem ser modificados para acompanhar as necessidades dos usuários.
2. Existem quatro tipos de manutenção, e a diferença está no foco de cada uma. Por exemplo: a manutenção corretiva é destinada a reparar os defeitos de um software existente; a manutenção preventiva

parte de uma observação reconhecida pelos mantenedores sobre o que poderá gerar algum tipo de erro no software, dessa forma tal erro será tratado antes que um problema venha a ocorrer; a manutenção adaptativa é necessária devido às necessidades de mudanças relacionadas a aspectos do ambiente do sistema, como o hardware, a plataforma do sistema operacional ou outro software de apoio que sofra uma mudança; a manutenção de aperfeiçoamento é necessária quando os requisitos de sistema mudam em resposta às mudanças organizacionais ou de negócio.

3. A reengenharia de software tem o objetivo de examinar os sistemas de informação e os aplicativos com a finalidade de reestruturá-los para que tenham melhor qualidade. Abrange uma série de atividades, como análise do sistema, engenharia reversa e reestruturação de programas e dados.
4. V – F – F – F – F – V – V – F – V – F

7. Gerenciamento de Configuração de Software

1. O gerenciamento de configuração de software é a atividade que identifica a configuração de um sistema em pontos distintos no tempo, com a finalidade de controlar sistematicamente as mudanças na configuração e manter a integridade e a rastreabilidade da configuração ao longo do ciclo de vida do software.**2.**
2. Um item de configuração de software é um elemento ou agregação de hardware ou software ou ambos projetado para ser gerenciado como um único elemento. Exemplo: plano do projeto, especificação e documentação de projeto, plano de testes, código-fonte, dicionário de dados, entre outros.
3. *Baseline* é uma linha de referência de software, ou seja, uma versão aprovada de um item de configuração, formalmente designada e fixa em um momento específico durante o ciclo de vida do item de configuração.
4. (1) Auditoria de configuração funcional do software: seu objetivo é garantir que o item de software auditado seja consistente com suas especificações de funcionamento.

(2) Auditoria de configuração física de software: seu objetivo é garantir que a documentação do projeto seja consistente com o produto de software construído.

8. Gerenciamento de Engenharia de Software

1. A primeira característica refere-se à intangibilidade, que dificulta a mensuração de atividades para planejamento e acompanhamento do progresso do projeto. A segunda característica é a variabilidade do escopo do projeto de software, que dificulta a análise de riscos até para os gerentes de projeto mais experientes. Por fim, as rápidas mudanças tecnológicas podem tornar obsoleta a experiência do gerente de projeto. Assim, as lições aprendidas de um projeto podem não servir para os futuros projetos.
2. Normalmente gerentes de projeto de software têm habilidades técnicas relacionadas às atividades de engenharia de software. No entanto, em relação às habilidades relacionadas aos aspectos humanos, que envolvem a capacidade de gerir conflitos e motivar a equipe na área de software, nem sempre os gestores estão preparados para a função.
3. D
Erros de escopo são os mais onerosos.
4. A principal diferença é o foco de cada prática. A Garantia da Qualidade visa à qualidade do processo de desenvolvimento, ou seja, o foco é a prevenção de erros. O Controle de Qualidade visa à detecção de erros nos artefatos do projeto para garantir que este esteja seguindo os procedimentos definidos pelas políticas de qualidade da organização.

9. Ferramentas, Métodos e Técnicas na Engenharia de Software

1. Ferramentas CASE são ferramentas automatizadas que têm como objetivo auxiliar o desenvolvedor de sistemas em uma ou várias

etapas do ciclo de desenvolvimento de software. As ferramentas CASE podem ser utilizadas em vários modos (INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, 2008):

- × como ferramentas autônomas – apenas a compatibilidade com elementos do ambiente deve ser abordada;
 - × em pequenos grupos que se comunicam diretamente uns com os outros – pode-se supor que a integração é predefinida;
 - × em grande escala – SEE (Software Engineering Environment) – a capacidade da ferramenta para utilizar os serviços relevantes da estrutura deve ser abordada; essa solução tem como premissa a existência de um conjunto integrado de ferramentas que agilizam e reduzem o esforço de trabalho, integrando e automatizando todas as fases do ciclo de vida de software.
2. Classificação das ferramentas CASE: Front end ou Upper CASE; Back end ou Lower CASE; I-CASE.
 3. Basicamente, as metodologias ágeis de desenvolvimento surgiram devido à constatação de que as abordagens tradicionais traziam muita burocracia a pequenos projetos de software.
 4. B
- Product Backlog.

10. Qualidade de Software

- 1.
- a) (V) O grande desafio da engenharia de software é descobrir as melhores técnicas e formas de se desenvolver softwares de qualidade.
 - b) (F) Testes de verificação fazem diferença, possibilitando encontrar defeitos nos levantamentos de requisitos, nos artefatos que forem ficando prontos, permitindo a entrega de um produto de software com boa qualidade.
 - c) (F) Existem muitos modelos disponíveis no mercado voltados para os processos de software, como ISO, CMM, MPS.BR, SPICE e outros.

- d) (F) Definir qualidade de software é algo bastante complexo, pois esse produto apresenta requisitos não implícitos que muitas vezes são desejados pelos clientes, tornando essa definição complicada e bastante subjetiva.
- e) (F) Muito pelo contrário, o software necessita de critérios para avaliar a sua qualidade, pois, sem eles, a avaliação fica na esfera da subjetividade, sem saber ao certo o que os clientes esperam ou desejam do produto.
- 2. Justamente por ser desenvolvido por pessoas, é necessário garantir que os profissionais envolvidos na produção de software estejam totalmente voltados para as práticas de qualidade, que não irão pular etapas para atender prazos, ou que irão codificar sem um padrão adequado, ou sem realizar testes unitários, de modo a passar o artefato para a próxima etapa com o mínimo de qualidade.
- 3. A afirmação está incorreta, pois a falta de qualidade também gera custos altos, os quais podem acarretar em produto encalhado ou até mesmo na perda da confiabilidade da empresa com seus clientes, gerando uma imagem negativa.
- 4. Não, há outras práticas, como auditorias, revisões técnicas, testes estáticos, entre outros.

Referências

ABRAN, P.; DUPUIS, R. (Ed.). **Guide to the Software Engineering Body of Knowledge: SWEBOK**. Los Alamitos: IEEE Computer Society, 2004.

ANDRIANO, N. **Proceso de elicitación de requerimientos para software empaquetado y software a medida**. Estadística. Universidad Nacional de Córdoba. In: 9º. WORKSHOP IBEROAMERICANO DE INGENIERÍA DE REQUISITOS Y AMBIENTES DE SOFTWARE (IDEAS), 2006, La Plata, Buenos Aires, Argentina, 2006.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **NBR ISO/IEC 12207**: tecnologia de informação: processos de ciclo de vida de software. Rio de Janeiro, 1998.

AVERSAN, L. et al. Managing Coordination and Cooperation in Distributed Software Process: The GENESIS Environment. **Software Process Improvement and Practice**, v. 9, p. 239-263, 2004.

BANAS QUALIDADE, ano XIII, n. 141, fev. 2004.

BARTIÉ, A. **Garantia de qualidade de software**: adquirindo maturidade organizacional. Rio de Janeiro: Campus, 2002.

BECK, K.; ANDRES, C. **Extreme programming explained**: embrace change. New Jersey: Addison-Wesley, 2001.

BOEHM, B. W. **A spiral model of software development and enhancement**. IEEE Computer Society Press Los Alamitos, California, v. 21, n. 5, 1988.

BROOKS, Fred P. **No Silver Bullet: Essence and Accident in Software Engineering**. Proceedings of the IFIP Tenth World Computing Conference: 1069–1076, 1986.

CAPGEMINI. **World Quality Report 2014 revela que investimentos em testes de aplicativos devem chegar a 29% em 2017**. 27 dez. 2014. Disponível em: <<https://www.br.capgemini.com/world-quality-report-2014-revela-que-investimentos-em-testes-de-aplicativos-devem-chegar-a-29-em>>. Acesso em: 16 set. 2016.

CARNEGIE MELLON UNIVERSITY. **Software Engineering Institute**. Disponível em: <<http://www.sei.cmu.edu>>. Acesso em: 20 set. 2016.

- CARVALHO, E. A. **Engenharia de processos de negócio e a engenharia de requisitos:** análise e comparações de abordagens e métodos de elicitação de requisitos de sistemas. Programa de Pós-Graduação em Engenharia de Produção. Rio de Janeiro: COPPE/UFRJ, 2009.
- DAVIS, A. M. **Operational prototyping:** a new development approach. University of Colorado, 1992.
- CHAOS REPORT (2015). Disponível em: <<https://www.projectsmart.co.uk/white-papers/chaos-report.pdf>>. Acesso em: 18 nov. 2016.
- CRESPO, A. N.; JINO, M. **Processo de teste de software.** Relatório Técnico. Cenpra: 2005.
- CROSBY, P. B. **Qualidade é investimento.** Rio de Janeiro: José Olympio, 1992.
- DAVIDSON, J. S. Reducing Risk in Software Projects Using Behavior-Based Requirements. **PM World Journal Reducing**, v. 1, n. 4, 2012. Disponível em: <www.pmworldjournal.net>. Acesso em: 3 abr. 2012.
- FOWLER, M. et al. **Refactoring:** improving the designer of existing code. Boston: Addison-Wesley, 1999.
- FUGGETTA, A. A classification of CASE Technology. **Journal Computer**, Los Alamitos, v. ed. 26, p. 25-38, 12 Dec. 1993.
- FURLAN, F. P. **Visualização de informação como apoio ao planejamento do teste de software.** Dissertação de Mestrado, Universidade Metodista de Piracicaba, Piracicaba, SP: 2009. Disponível em: <<https://www.unimep.br/phpg/bibdig/pdfs/2006/PIICEDIGLYDU.pdf>>. Acesso em: 22 out. 2016
- FURLAN, J. D. **Modelagem de objetos através da UML:** Análise e desenho orientado a objetos. São Paulo: Makron Books, 1998.
- GUEDES, G. T. A. **UML 2:** uma abordagem prática. 2. ed. São Paulo: Novatec, 2011.
- HAKIN, J.; SPITZER, T. **Effective prototyping for usability.** In: ANNUAL CONFERENCE ON COMPUTER DOCUMENTATION. 18., 2000, Massachusetts. **Proceedings.** IEEE, 2000. p. 47-54.

- HASS, A. M. J. **Configuration management principles and practices.** Indianapolis: Addison-Wesley Professional, 2013.
- IEEE. The Institute of Electrical and Electronics Engineering. **Guide to the Software Engineering Body of Knowledge:** SWEBOK. A Project of the IEEE Computer Society, v. 4. 2004.
- _____. **Software**, v. 9, n. 5, p. 70-78, 1992.
- _____. Std 830-1998 IEEE. **Recommended practice for software requirements specifications**, IEEE Computer Society, 1998.
- INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. **Std. 828-2012:** IEEE Standard for Configuration Management in Systems and Software Engineering. Piscataway, 2012.
- INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. **ISO/IEC 14102:2008:** Information technology: Guideline for the evaluation and selection of CASE tools. Geneva, 2008.
- _____. **ISO/IEC/IEEE 24765:** Systems and Software Engineering: Vocabulary. Geneva, 2010.
- KOTONYA, P.; SOMMERVILLE, I. **Requirements engineering:** processes and techniques. Wiley, 1998.
- KROGSTIE, J.; JAHR, A.; SJOBERG, D. I. K. A longitudinal study of development and maintenance. In Norway: Report from the 2003. **Information and software technology**, v. 48, n. 11, 2005, p. 993-1005.
- KRUCHTEN, P. **The rational unified process:** an Introduction. 3. ed. New Jersey: Addison Wesley, 2003.
- LEET, K. M.; UANG, C-M.; GILBERT, A. M. **Fundamentos da análise estrutural.** 3. ed. Porto Alegre: AMGH, 2010.
- LEITE, J. C. S. P. **Engenharia de requisitos.** 1994. Disponível em: <<http://livrodeengenhariaderequisitos.googlepages.com/ERNOTASDEAULA.pdf>>. Acesso em: 20 nov. 2016.
- LIENTZ, B. P.; SWANSON, E. B. **Software maintenance management.** Reading, mass: Addison-wesley, 1980.

- LOBO, E. J. R. **Guia prático de engenharia de software**. São Paulo: Digerati Books, 2009.
- LOWDERMILK, T. **User-Centered Design**. O'Reilly Media, 2013.
- MALDONADO, J. C. **Critérios potenciais usos**: uma contribuição ao teste estrutural de software. Campinas: DCA/FEE/UNICAMP, 1998.
- MARTIN, M. V.; ISHII, K. Design for Variety: Developing Standardized and Modularized
- MCCONNELL S. **Code Complete**. 2. ed. Microsoft Press, 2004.
- MCMURTREY, M. E. et al. Current utilization of CASE technology: lessons from the field. **Industrial Management & Data Systems**, Bingley, v. 100, n. 1, p. 22-30, 2000.
- MEIRELLES, P. R. **Teste integrado de software e hardware**: reusando casos de teste de software em teste de microprocessadores. Dissertação de Mestrado, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2008. Disponível em: <<http://www.lume.ufrgs.br/bitstream/handle/10183/25520/000751158.pdf>>. Acesso em: 18 out. 2016.
- Modularization During Concept Development Phase. **International Journal of Production Economics**, v. 96, n. 2, 2005.
- MOULE, J. **Killer UX Design**: SitePoint Pty, Limited. 2012.
- NEPAL, B.; MONPLAISIR, L.; SINGH, N. Integrated Fuzzy Logic-Based Model for Product
- NUSEIBEH, B.; EASTERBROOK, S. Requirements engineering: a roadmap. In: CONFERENCE ON THE FUTURE OF SOFTWARE ENGINEERING. **Proceedings of the...** Limerick, Ireland, 2000.
- PFLEEGER, S. L. **Engenharia de software**: teoria e prática. 2. ed. São Paulo: Prentice Hall, 2004.
- OCVIRK, O. C. et al. **Fundamentos de arte**: teoria e prática. 12. ed. Porto Alegre: AMGH Editora, 2014.

OLIVEIRA, E. A. et al. **Um modelo de fórum de discussão para representações fidedignas de ideias.** In: Anais do simpósio brasileiro de informática na educação, 2009

OLIVEIRA, O. J. **Gestão da qualidade:** tópicos avançados. São Paulo: Pioneira Thomson, 2003.

PAULA FILHO, W. de P. **Engenharia de Software:** fundamentos, métodos e padrões. 2. ed. Rio de Janeiro: LTC, 2005.

PESSOA, M. S. CMM. In: ROCHA, A. R. C. da; MALDONADO, J. C.; WEBER, K. C. (Org.). **Software:** teoria e prática. São Paulo: Prentice Hall, 2001.

PEZZÈ, M.; YOUNG, M. **Teste e análise de software:** processos, princípios e técnicas. Porto Alegre: Bookman, 2008.

PFLEEGER, S. L. **Engenharia de software:** teoria e prática. 2. ed. São Paulo: Prentice Hall, 2004.

PRESSMAN, R. S. **Engenharia de software.** 7. ed. São Paulo: Makron Books, 2011.

PRESSMAN, R. S. **Engenharia de software:** uma abordagem profissional. 7. ed. Columbus, McGraw-Hill, 2011.

PRESSMAN, R.; MAXIM, B. R. **Engenharia de software:** uma abordagem profissional. São Paulo: Makron Books, 2016.

PRODUCT Platform Architectures. **Research in Engineering Design**, Berlin, v. 13, p. 213-235, 2002.

PROJECT MANAGEMENT INSTITUTE, I. **Um guia do conhecimento em gerenciamento de projetos (guia pmbok®).** 5. ed. Newtown Square, Pennsylvania: Saraiva, 2013.

QUALITY WAY. **David Garvin e as oito dimensões da qualidade.** Disponível em: <<https://qualityway.wordpress.com/2015/08/18/david-a-garvin-e-as-oito-dimensoes-da-qualidade-por-gregorio-suarez-parte-1/>>. Acesso em: 15 out. 2016.

REZENDE, D. A. **Engenharia de software e sistemas de informações.** Rio de Janeiro: Brasport, 2005.

- RIOS, A. B. E.; CRISTALLI, R.; MOREIRA, T. **Base de conhecimento em testes de software**. 3. ed. São Paulo: Martins Fontes, 2012.
- RIOS, E. **Documentação de Teste**: dissecando a norma ou padrão IEEE 829-2008. 2008. Disponível em: <http://www.emersonrios.eti.br/teste%20artigos_arquivos/Documentacao%20de%20teste%202%20horas.pdf>. Acesso em: 20 set. 2016.
- RIOS, E.; MOREIRA, T. **Teste de software**. 3. ed. Rio de Janeiro: Alta Books, 2013.
- ROBERTSON, J.; VOLERE S. **Requirements specification template**, Edition 14, 2009. Disponível em: <<http://www.volere.co.uk>>. Acesso em: 20 jan. 2017.
- ROCHA, A. R. C. da. **Qualidade de software – Teoria e prática**. São Paulo: Prentice Hall, 2001. UNICRUZ. Universidade de Cruz Alta, Cruz Alta. Disponível em: <<http://dinf.unicruz.edu.br/~facco/projeto.htm>>. Acesso em: 20 set. 2016.
- ROCHA, A. R. C. et al. **Qualidade de software: teoria e prática**. São Paulo: Prentice Hall, 2001.
- ROCHA, A. R. C.; MALDONADO, J. C.; Weber, K. C. **Qualidade de software: teoria e prática**. São Paulo: Prentice Hall, 2001.
- SALVIANO, C. et al. In: ROCHA, A. R. C.; MALDONADO, J. C.; WEBER, K. C. (eds) **Qualidade de software: teoria e prática**. São Paulo, Prentice Hall, 2001.
- SALVIANO, C. F. **Melhoria e avaliação de processo com ISO/IEC 15504 e CMMI**. Lavras: UFLA, 2003.
- SAND, J. C.; GU, P.; WATSON, G. Home: house of modular enhancement: a Tool for Modular Product Redesign. **Concurrent Engineering**, v. 10, n. 2, 2002.
- SANTANA JÚNIOR, C. A. de. **Avaliação da utilização de melhoria de processo de software baseada em metodologias ágeis em empresas CMMI**, 2012.

SHORE, J. These 10 Peter Drucker Quotes May Change Your World. 2014. Disponível em: <<https://www.entrepreneur.com/article/237484>>. Acesso em: 20 nov. 2016.

SILVA, M. da. Desenvolvimento de uma ferramenta de testes para um sistema integrado em software e hardware. Universidade Federal de Lavras, Lavras, MG: 2005.

SILVA, U. R. Avaliação e desenvolvimento da equipe do projeto. **Techoje – RH, Liderança, Comunicação.** Disponível em: <http://www.techoje.com.br/site/techoje/categoria/detalhe_artigo/47>. Acesso em: 18 nov. 2016.

SIMPSON, T. W. et al. From User Requirements to Commonality Specifications: an Integrated Approach to Product Family Design. **Research in Engineering Design**, Berlin, v. 23, 2012.

SOMMERVILLE, I. **Engenharia de software.** 8. ed. São Paulo: Addison Wesley Bra, 2007.

_____. **Engenharia de software.** 9. ed. São Paulo: Pearson Prentice Hall, 2011.

SOWUNMI, O. Y.; MISRA, S. An Empirical Evaluation of Software Quality Assurance Practices and Challenges in a Developing Country. In: 2015 IEEE INTERNATIONAL CONFERENCE ON COMPUTER AND INFORMATION TECHNOLOGY; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing.

STAA, A. **Programação modular.** Rio de Janeiro: Campus, 2000.

SWEBOK. Guide for the Software Engineering Body of Knowledge. 2004 version, IEEE Computer Society, California, EUA.

_____. **Guide of the Software Engineering Body of Knowledge.** 2016. Disponível em: <<http://www.swebok.org>>. Acesso em: 14 out. 2016.

THEBEAU, R. E. **Knowledge Management of System Interfaces and Interactions for Product Development Processes.** 149 f. Tese (Mestrado em Engenharia e Gestão) – Massachusetts Institute of Technology, Massachusetts, 2001.

- VIANNA, M.; et al. **Design Thinking**. Rio de Janeiro: MJV Press, 2012.
- WAZLAWICK, R. S. **Engenharia de software**: conceitos e práticas. Rio de Janeiro: Elsevier, 2013.
- WEBER, K. C.; ROCHA, A. R. C. da. NASCIMENTO, C. J. do. (Org.). **Qualidade e produtividade em software**. São Paulo: Makron Books, 2001.
- WORLD Quality Report. 6th ed. [S.l.]: Capgemini; Sogeti; HP, 2014.
- XAVIER, J. H. In: WEBER, K. C. et al. **Qualidade e produtividade em software**, São Paulo, Ed. Makron Books, 2001.
- YOUNG, R. R. **Effective requirements practices**. 2. ed. Addison-Wesley Information Technology Series, 2002.
- YOURDON, E. **Análise estruturada moderna**. [São Paulo]: Campus, 1992.

O avanço tecnológico do século XX trouxe diversos desafios para a área de Tecnologia da Informação. Nesse cenário, surge a necessidade de se construir, gerenciar e manter sistemas e programas computacionais de forma inovadora, com qualidade e eficiência.

A partir de uma abordagem atualizada, este livro foi estruturado para apresentar conceitos, processos e características de um sistema, apontando a importância dessas etapas para o controle de todo o seu ciclo de vida.

Para garantir a completa síntese do assunto, a obra também expõe algumas habilidades essenciais para o profissional que deseja atuar no ramo de Engenharia de Software, visando o melhor desempenho das atividades exigidas no mercado de trabalho.

