

# ENGENHARIA DE SOFTWARE

Qualidade e Produtividade com Tecnologia

KECHI HIRAMA



Preencha a **ficha de cadastro** no final deste livro e receba gratuitamente informações sobre os lançamentos e promoções da Elsevier.

Consulte também nosso catálogo completo, últimos lançamentos e serviços exclusivos no site  
**[www.elsevier.com.br](http://www.elsevier.com.br)**

# ENGENHARIA DE SOFTWARE

Qualidade e Produtividade com Tecnologia

KECHI HIRAMA



© 2012, Elsevier Editora Ltda.

Todos os direitos reservados e protegidos pela Lei nº 9.610, de 19/02/1998.

Nenhuma parte deste livro, sem autorização prévia por escrito da editora, poderá ser reproduzida ou transmitida, sejam quais forem os meios empregados: eletrônicos, mecânicos, fotográficos, gravação ou quaisquer outros.

Coordenação Editorial: Letícia Vendrame

Preparação de Texto: Renata Mendonça

Revisão: Alexandra Resende

Editoração Eletrônica: Tony Rodrigues

Elsevier Editora Ltda.

Conhecimento sem Fronteiras

Rua Sete de Setembro, 111 – 16º andar

20050-006 – Rio de Janeiro – RJ

Rua Quintana, 753 – 8º andar

04569-011 – Brooklin – São Paulo – SP

Serviço de Atendimento ao Cliente

0800 026 53 40

sac@elsevier.com.br

ISBN: 978-85-352-4882-1

**Nota:** Muito zelo e técnica foram empregados na edição desta obra. No entanto, podem ocorrer erros de digitação, impressão ou dúvida conceitual. Em qualquer das hipóteses, solicitamos a comunicação à nossa Central de Atendimento, para que possamos esclarecer ou encaminhar a questão.

Nem a editora nem o autor assumem qualquer responsabilidade por eventuais danos ou perdas a pessoas ou bens, originados do uso desta publicação.

CIP-BRASIL. CATALOGAÇÃO-NA-FONTE  
SINDICATO NACIONAL DOS EDITORES DE LIVROS, RJ

---

H559e

Hirama, Kechi

Engenharia de software : qualidade e produtividade com tecnologia / Kechi

Hirama. - Rio de Janeiro : Elsevier, 2011.

Inclui bibliografia

ISBN 978-85-352-4882-1

1. Engenharia de software. 2. Software - Desenvolvimento. I. Título.

---

11-5976.

CDD: 005.1 CDU: 004.41

---



## DEDICATÓRIA

---

Dedico este livro aos meus pais Toshio e Hatuio (*in memoriam*)  
e aos meus filhos Yukari e Jun pelos anos de felicidade  
que passamos e pelos que ainda virão.

## AGRADECIMENTOS

---

A Universidade de São Paulo (USP) que me possibilita trabalhar em uma entidade de referência de ensino superior no Brasil.

À Escola Politécnica da Universidade de São Paulo (Epusp) que me possibilita trabalhar em um ambiente de excelência tecnológica.

Ao Departamento de Engenharia de Computação e Sistemas Digitais – PCS da Epusp – que me possibilita trabalhar em um ambiente de pioneirismo, que teve a sua origem com o Laboratório de Sistemas Digitais (LSD), onde nasceu o primeiro computador digital brasileiro em 1972.

Ao Laboratório de Tecnologia de Software (LTS) que me possibilita trabalhar no desenvolvimento da área de Engenharia de Software e, em particular, à Profa. Dra. Selma Shin Shimizu Melnikoff pelos longos anos de amizade e apoio.

Ao Grupo de Sistemas Complexos (GSC) que é jovem ainda, e me desperta novas ideias e desafios nas pesquisas em sistemas computacionais complexos.

Aos meus orientados de mestrado e de especialização pela troca de ideias e trabalhos concluídos e aos meus alunos de graduação das disciplinas Fundamentos de Engenharia de Software, Fundamentos de Sistemas de Tempo Real e Gerência, Qualidade e Tecnologia de Software; aos de pós-graduação da disciplina Qualidade de Software: Conceitos e Paradigmas; e aos de extensão do curso de MBA – Tecnologia de Software das disciplinas Qualidade de Software e Projeto Integrado pelas discussões em salas de aula.

Aos funcionários do PCS pelo apoio e pelo ambiente amigável e descontraído.

Aos meus irmãos (Morio e Tsuyako), cunhados (Eidi e Massako) e sobrinhos (Daigo, Dandy, Marcelo e Henrique) que sempre me acompanharam nesta caminhada.

*“Somos o que repetidamente fazemos.  
A excelência, portanto, não é um feito, mas um hábito.”  
(Aristóteles)*



## PREFÁCIO

---

Sou engenheiro eletricista de formação com ênfase em sistemas digitais. Desde o início da minha carreira, aplicar conceitos e técnicas aprendidas era a forma de me ver como um verdadeiro engenheiro, sempre balizando a solução pelo seu custo/benefício. Paralelamente, percebi que a tecnologia a cada dois anos mudava muito e em seguida, a cada um ano. Atualmente, além do ciclo vida das tecnologias ser menor, a tecnologia é muito mais variada e integrada.

Como me formei no final da década de 1970, tive uma formação mais densa em hardware. O software invadiu o espaço à medida que o hardware evoluiu e o computador foi chamado para processar e controlar várias atividades de nossa vida, como produzir um veículo ou editar um texto. Portanto, o hardware deveria ser cada vez mais versátil para dar conta de novas aplicações. A evolução do hardware foi caracterizada por processadores mais potentes e memórias para armazenamento de dados de maior capacidade e mais baratos.

Naquela época, o software era um componente menor do computador. A diversidade de possibilidades de uso do computador fez com que fossem tomadas decisões de padronização de uma plataforma e as aplicações foram desenvolvidas em software. Porém, este ainda era desenvolvido como o hardware até ter o *status* de um produto de engenharia.

Entre as engenharias tradicionais, a engenharia de software é particular. Seu produto é novo e, muitas vezes, se começa do zero, desenvolve-se

inclusive os seus componentes. Além disso, necessita-se trabalhar com diversos níveis de abstração para poder dar conta de sua complexidade e, por fim, chegar a um software que atenda às necessidades do cliente. Todo produto de software é fruto de um projeto. Assim, o projeto deve ser gerenciado para não comprometer os prazos e custos acordados com o cliente.

Os primeiros softwares eram voltados para aplicações comuns de folha de pagamentos e controle de estoques. Atualmente, os softwares são fatores de competitividade e os negócios não sobrevivem sem um apoio de softwares de qualidade. Uma falha ocorrida em uma transação financeira pode ocasionar muitas perdas aos clientes, uma falha na ativação de um motor reserva pode ocasionar a explosão de uma usina nuclear, uma falha em uma interface pode levar um piloto a tomar decisões erradas que levem à queda do avião.

Um desafio constante é como desenvolver softwares cada vez maiores e integrados com outros softwares e, ainda, ter qualidade e produtividade adequada para atender às necessidades dos clientes e cumprir os prazos e custos.

Este livro é um relato de uma trajetória vivida durante os meus 30 anos de carreira. Destes, 15 anos foram dedicados a dois grandes sistemas de automação e controle em laboratório e centro de pesquisa, e os outros 15 foram dedicados a ensino, pesquisa e extensão e consultoria a empresas na área de Tecnologia da Informação (TI). Por considerá-la muito rica, decidi colocar no papel uma série de aprendizados e experiências em projetos de software.

Durante este longo período, tive influências de grandes autores de Engenharia de Software, tais como Ian Sommerville, Roger S. Pressman e Shari Lawrence Pfleeger, nessa ordem. Suas obras direcionaram minha carreira como engenheiro de software e como professor, pesquisador e consultor, e certamente, as ideias para este livro.

Tentei escrever um livro mais leve, de fácil leitura e manipulação do que os de minhas influências. A intenção foi escrever algo mais adequado aos alunos de graduação e aos professores em sala de aula dos cursos de

Engenharia de Computação, Sistemas de Informação e Ciência da Computação. Como pode ser visto na estrutura do livro, considero os temas discutidos fundamentais para um bom projeto de software.

Pretendo neste livro mostrar conceitos e técnicas que possibilitem dar aos projetos de software qualidade e produtividade. Assim, este livro poderá ser útil não somente para cursos de graduação, mas também para cursos de pós-graduação e para as empresas que atuam na área de TI.


Os alunos de pós-graduação poderão encontrar discussões para suas pesquisas. Os desenvolvedores das áreas de TI das empresas terão a possibilidade de reciclar suas visões de projeto de software e estarem mais preparados para lidar com os problemas do dia a dia.

Aos leitores iniciantes na área de Engenharia de Software, recomendo a leitura desde o início, sem pular as seções dos capítulos. Os leitores mais familiarizados com a área podem ir aos temas de interesse, pois as seções são pouco acopladas. Em ambos os casos, recomendo fortemente que façam os exercícios propostos.

Kechi Hirama

# Visão geral

---

 software está presente em todas as áreas do cotidiano, seja controlando, entretendo, vigiando, comunicando etc. Então, pode-se considerar o software como um componente fundamental do dia a dia. Ele precisa funcionar, dar respostas precisas, ser rápido e barato. Como seria possível conceber e produzir algo tão especial? Existem diversas formas. Uma delas é contratar profissionais com grande experiência na aplicação para desenvolver o software. A outra, mais permanente, é implantar processos, métodos e ferramentas para que o desenvolvimento seja mais previsível independente de quem produza o software. Neste contexto, a disciplina de Engenharia de Software desempenha um papel fundamental para desenvolver software com qualidade e produtividade.

---

## 1.1. INTRODUÇÃO

*Um estudo recente apontou a Tecnologia da Informação (TI) como sendo uma das áreas de trabalho mais estressantes. Exigem-se dos profissionais que desenvolvem software de TI capacidade cognitiva para gerar soluções aderentes ao negócio, de cumprimento de prazos e custos rigorosos etc. Não bastasse isso, ainda têm de gerenciar pressões internas e externas. Assim, os softwares são em geral liberados sem a qualidade desejada e são melhorados após várias manutenções, corrigindo-se os defeitos. Para minimizar essa situação, a disciplina de Engenharia de Software tem contribuído muito através de ferramentas, métodos e processos.*

O software desempenha um papel fundamental em várias áreas de negócios seja para controle e segurança de informações, controle de processos, apoio a decisões como, também, entretenimento. Há 40 anos, o software começava a se tornar complexo e com isso, problemas de qualidade e atendimento de prazos e custos já eram apontados. Assim, nasceu a Engenharia de Software tratando o software como um produto de engenharia para fazer frente aos novos desafios.

A Engenharia de Software abrange ferramentas de apoio para as atividades, métodos para orientar a realização das atividades, processo para definir as atividades e os produtos e a qualidade de processo e de produto de software. Dessa maneira o desenvolvimento de software pode obter produtos com qualidade e produtividade.

Métodos de desenvolvimento de software têm sido propostos desde a década de 1970. Os métodos mais conhecidos foram baseados em duas abordagens: Estruturada e Orientada a Objetos. Ambas possibilitam desenvolver sistemas em várias áreas de aplicação. Atualmente, os métodos que seguem a abordagem Orientada a Objetos têm maior aceitação devido à notação UML (*Unified Modeling Language*), com o apoio da OMG (*Object Management Group*), ter se tornado um padrão da indústria para o desenvolvimento de software.

Nessa linha, processos de software são importantes, pois estabelecem para os membros da equipe de projeto uma diretriz de o que deve ser feito para atender aos objetivos do software. Ou seja, eles definem quais são as atividades que permitem obter o produto de software. O processo Cascata (em inglês, *Waterfall Model*), proposto na década de 1970, reflete o que é mais usado em outros projetos de engenharia e é, ainda, muito usado no desenvolvimento de software, particularmente quando faz parte de um projeto maior de engenharia de sistemas. Outros processos também foram propostos, tais como o evolutivo, o modelo V, processo unificado (RUP – *Rational Unified Process* – IBM/Rational) e, mais recentemente, o *Extreme Programming* (XP).

Sendo o software um produto de engenharia, ele deve ser tratado como resultado de um projeto. Assim, a atividade de gerenciamento é fundamental para atingir os resultados desejados, dentro de restrições de custo

e prazo. O gerenciamento de requisitos trata da entrega dos requisitos acordados e das mudanças de requisitos ao longo do ciclo de desenvolvimento de software.

As atividades de desenvolvimento de software abrangem essencialmente atividades técnicas de Engenharia de Sistemas, análise, projeto, codificação e testes. A atividade de Engenharia de Sistemas tem por objetivo entender as necessidades de negócio do cliente e especificar os requisitos do sistema computacional, incluindo-se os requisitos do software em alto nível. A atividade de Análise busca entender os requisitos do sistema e detalhar os requisitos do software. A atividade de Projeto tem por objetivo estabelecer uma arquitetura do software que seja executável. A atividade de Codificação pretende traduzir as especificações de software em códigos de programa que sejam processados por um sistema computacional. A atividade de Testes tem por objetivo descobrir defeitos no software, considerando aspectos estruturais e funcionais do software.

Atualmente, muitas aplicações são voltadas para usuários que dialogam e obtêm informações dos sistemas computacionais. Uma interface com o usuário permite a comunicação entre o usuário e o sistema computacional. Assim, o projeto de interface exige conhecimentos de fatores humanos e tecnologia de interfaces.

Todo projeto de software, além de bem gerenciado e desenvolvido, precisa garantir a qualidade do produto por meio de técnicas de verificação e validação, gerenciamento de configuração e de qualidade.

Os processos de desenvolvimento de software exigem a aplicação de conceitos e técnicas seguindo métodos de desenvolvimento. As ferramentas CASE surgiram para apoiar as atividades de desenvolvimento proporcionando produtividade e qualidade do software.

Novas tecnologias estão sendo pesquisadas e algumas delas já são de uso corrente na indústria de software. A tecnologia de Reuso é uma forma de aumentar a produtividade com o reuso de sistemas ou componentes existentes. Isso ocorre na maioria das disciplinas de engenharia, mas na Engenharia de Software o reuso sistemático de software é ainda incipiente.

A tecnologia de Orientação a Serviços (SOA – *Service Oriented Architecture*) torna possível, além do reuso de aplicações, o acesso a informações diversas disponíveis em vários computadores ligados à internet. A tecnologia de Orientação a Aspectos permite tratar mais adequadamente os requisitos não funcionais do software.

Para atingir índices cada vez mais altos de qualidade e produtividade é fundamental para uma organização de software adotar padrões e modelos de qualidade. Um dos modelos mais difundidos atualmente é o CMMI (*Capability Maturity Model Integration*) que é um modelo de maturidade de melhoria contínua de processo, para o desenvolvimento de produtos e serviços do SEI (*Software Engineering Institute*) da Carnegie Mellon University dos Estados Unidos. O CMMI tem reconhecimento mundial, e no Brasil existem muitas empresas que possuem certificações baseadas neste modelo. O outro modelo, o Modelo de Referência para Melhoria do Processo de Software (MR-MPS) faz parte do Programa MPS-BR que é uma iniciativa brasileira para melhoria contínua de processos de software. O MR-MPS tem sido muito aceito ultimamente no Brasil e existem muitas empresas certificadas também neste modelo.

### **O que se pretende com este livro**

O objetivo deste livro é apresentar e refletir sobre as principais abordagens de desenvolvimento de software, destacando-se as suas atividades e, sobretudo, os fundamentos de Engenharia de Software.

Assim, o livro é dividido em 12 capítulos: Visão Geral, Processos de Software, Gerenciamento, Desenvolvimento de Software, Desenvolvimento de IHM, Verificação e Validação, Configuração, Qualidade, Ferramentas CASE, Tecnologias, Modelos de Maturidade e Considerações Finais.

As seções dos capítulos descrevem a essência de um tema e, em seguida, são apresentados alguns comentários sobre o tema. Portanto, cada seção não é muito detalhada. Em cada uma delas, são propostos 10 exercícios que podem ser desenvolvidos individualmente pelos alunos para ampliar e aprofundar seus conhecimentos ou em salas de aula pelo professor. Em seguida,

apresento uma lista de leituras que considero relevantes sobre o tema.

No final do livro, são apresentados um glossário sobre alguns termos relevantes, as referências usadas neste livro e uma bibliografia complementar.

Espera-se que os alunos compreendam que o desenvolvimento de software deve seguir uma disciplina apoiada nos conceitos e técnicas de Engenharia de Software para obter produtos com qualidade e produtividade.

No entanto, para a aplicação adequada dos conceitos e técnicas discutidos, investimentos adequados em processos, pessoas e tecnologias devem ser feitos para trazer benefícios para a Organização.

### **Questões mais frequentes (FAQ)**

- **Software é somente programa de computador?**

Em certo sentido, sim. Mas não é tudo. Na Engenharia de Software consideram-se “softwares” os artefatos resultantes das atividades de levantamento de requisitos, análise de requisitos, projeto, codificação, testes e outras dentro do seu ciclo de vida. Assim, são considerados também todos os modelos e documentos produzidos.

- **Por que é necessário ter uma disciplina de Engenharia de Software? Não é suficiente somente saber programar?**

Desenvolver e manter software são muito dispendiosos. Muitos estudos indicam que para reduzir os custos e entregar o software nos prazos contratados com o cliente, há que se ter uma disciplina para ter sucesso com desenvolvimento de software.

- **Com tantas novas tecnologias de software surgindo a cada momento, seria possível dominar todas elas?**

Certamente, não. Embora elas surjam com frequência, o importante é que o desenvolvedor domine os conceitos e as técnicas fundamentais de como obter um software com qualidade e produtividade. Nesse contexto, a Engenharia de Software desempenha um papel fundamental.

- **As ferramentas de software disponíveis no mercado não resolvem o problema de falta de qualidade e produtividade em desenvolvimento de software?**



Ajudam, mas não resolvem. Sem dúvida, uma boa ferramenta é sempre bem-vinda. No entanto, ao basear-se somente nela para cobrir deficiências de conceitos e técnicas dos desenvolvedores, o software vai certamente custar mais caro e sair sem qualidade. Uma ferramenta é apenas um instrumento para um desenvolvedor já capacitado.

- **Quais são os efeitos da aplicação de conceitos e técnicas de Engenharia de Software na manutenção?**

Os custos de manutenção crescem exponencialmente em relação aos custos de desenvolvimento caso o software tenha sido desenvolvido sem os conceitos e as técnicas adequados.

- **Por que o desenvolvimento de software apoiado na Engenharia de Software exige muita documentação?**

É possível conduzir o desenvolvimento de software como um projeto de engenharia ou apenas como a criação de um programa. Embora, o software possa funcionar conforme o esperado no início, no primeiro caso o software deve passar por muitas manutenções e ainda continuar a ser útil, no segundo, não é possível prever. O que torna o primeiro caso melhor é que as decisões de projeto ficam registradas em documentos. Portanto, não é a quantidade, mas a importância da documentação para o ciclo de vida do software que prevalece.

- **Se o software vai ser alterado constantemente, vale a pena manter os seus documentos atualizados?**

A atualização da documentação é necessária para saber em qualquer tempo como o software evoluiu. Ajuda na estimativa de esforço e análise de impactos das futuras alterações.

- **Quais conceitos e técnicas de Engenharia de Software são úteis para um desenvolvimento de software?**

Os conceitos e as técnicas de Engenharia de Software têm evoluído ao longo dos anos. Uma boa maneira é definir conceitos e técnicas a partir de um processo de desenvolvimento estabelecido. Os processos são menos mutáveis do que conceitos e técnicas.

## 1.2. ENGENHARIA DE SOFTWARE

*Após 40 anos desde a sua definição, a Engenharia de Software evoluiu e tem apoiado o desenvolvimento de software em diversas áreas de aplicação. Nos dias atuais, onde o negócio sofre mudanças constantes, o software desempenha um papel fundamental. Nesse contexto, é necessário que o software seja aderente ao negócio de tal maneira que as organizações possam ser competitivas. A Engenharia de Software abrange ferramentas de apoio para atividades, métodos para orientar a realização das atividades, processo para definir as atividades e a qualidade dos processos e dos produtos de software. Dessa maneira, a Engenharia de Software pode obter produtos com qualidade e produtividade.*

O conceito “Engenharia de Software” foi cunhado em 1969 por Fritz Bauer em uma conferência patrocinada por um Comitê de Ciência da Organização do Tratado do Atlântico Norte (Otan),<sup>1</sup> no momento em que a chamada *crise do software* precisava de uma solução para a demanda crescente por software, dentro de custo e prazo adequados.

Nessa época, a crise foi identificada pela preocupação crescente na comunidade de software com a quantidade de defeitos, entregas fora de prazo e altos custos do software.

De acordo com vários relatos, Fritz Bauer teria dito:

*A Engenharia de Software é o estabelecimento e uso de sólidos princípios de engenharia a fim de obter um software que seja confiável e que funcione de forma econômica e eficiente em máquinas reais.*

É importante notar que a Engenharia de Software segue os mesmos princípios de uma disciplina de engenharia tradicional, baseada em uma relação adequada de custo/benefício do produto, que não falhe e que seja eficiente.

Outras definições podem ser encontradas nas normas da IEEE (*Institute of Electrical and Electronics Engineers*) dos Estados Unidos, onde a Engenharia de Software é:<sup>2</sup>

- a) a aplicação de uma abordagem sistemática, disciplinada e quantificável para o desenvolvimento, a operação e manutenção de software;
- b) o estudo de abordagens como descrito em (a).

Fritz Bauer<sup>1</sup> também declarou que *os sistemas deveriam ser construídos em níveis e em módulos*, que formam uma estrutura matemática. De certa maneira, a Engenharia de Software tem evoluído nesta linha e, atualmente, segundo Pressman, ela pode ser mais bem entendida como uma tecnologia em camadas ou níveis, conforme pode ser vista na Figura 1.1.



**Figura 1.1** – Camadas da Engenharia de Software.<sup>3</sup>

Na base da Figura 1.1, formando a camada foco em qualidade, dá-se ênfase à preocupação de qualquer disciplina de engenharia que é a qualidade. A qualidade na Engenharia de Software é baseada nos conceitos de gerenciamento da qualidade total (TQM – *Total Quality Management*) para a melhoria contínua dos processos.<sup>4,5</sup> O TQM é uma abordagem de gerenciamento organizacional (princípios, métodos e técnicas) para obter sucesso em longo prazo através da satisfação dos clientes.

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

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

A camada de ferramentas provê apoio automatizado ou semiautomatizado para processos e métodos. As ferramentas da área de Engenharia de Software são conhecidas como CASE (Engenharia de Software Apoiada por Computador do inglês *Computer-Aided Software Engineering*), que será discutida no Capítulo 9).

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

A disciplina de Engenharia de Software é muito abrangente. Processos, métodos, técnicas e ferramentas têm sido propostos com frequência. Esta contínua evolução da Engenharia de Software aponta para a necessidade de melhorar as abordagens atuais para produzir softwares cada vez mais complexos com qualidade e produtividade.

No entanto, existe uma distância entre o que é proposto na Engenharia de Software e o que é aplicado nos desenvolvimentos de software. Muitas organizações já detectaram que precisam formalizar as atividades de desenvolvimento, sem a qual estariam perdendo o controle do software. Isso também pode ser visto quando se contabiliza o número de retrabalhos durante e após o desenvolvimento do software por conta de inúmeros defeitos.

Se por um lado, há um compromisso com o cliente em relação aos prazos e custos, também as expectativas do cliente não são plenamente atendidas, ou seja, nem toda a funcionalidade do software contratada é entregue ao cliente.

Segundo o *Chaos Report* de 2009 da empresa norte-americana *Standish Group*, o percentual de sucesso dos projetos de software foi de 32%, de falha foi de 24% e de deficientes foi de 44%. Sucesso significa projeto entregue dentro das expectativas do cliente em relação a custo, prazo e atendimento. Falha significa projeto não concluído ou abandonado. E entre os deficientes estavam 45% acima do custo, 63% acima do prazo e 67% de expectativas atendidas.<sup>6</sup>

Entre as causas estavam requisitos de software mal definidos, incompletos ou alterados, falta de preparo da gerência, falta de conhecimento de tecnologia e problemas de comunicação.

Estes números tornam o apoio da disciplina de Engenharia de Software necessário para o sucesso dos projetos de software.

Existem várias abordagens para se desenvolver um software. No entanto, antes de tudo, exige-se o domínio dos fundamentos de Engenharia de Software.

Nesta seção, destacou-se o papel da Engenharia de Software em apoiar o desenvolvimento de software. No entanto, conforme foi apresentado, processos e ferramentas não são suficientes sem as pessoas inseridas nos processos. Outros pontos importantes a serem pensados são: Como se poderiam estimar os custos e os prazos de um projeto de software sem processos? Se os projetos de software em geral já chegam com custos e prazos inadequados, como se poderia obter qualidade em software? Como se poderia argumentar sobre outros parâmetros, tais como recursos necessários? A resposta está em um processo efetivo dentro de uma organização.

## EXERCÍCIOS

1. Os conceitos de Engenharia de Software são uma realidade nas organizações?
2. Quais são os fatores que dificultam a adoção de boas práticas de Engenharia de Software?

3. É possível obter qualidade de software sem um processo ou método de desenvolvimento?
4. Segundo as pesquisas da empresa *Standish Group*, os percentuais de desempenho de projetos de software têm se mantido ao longo de vários anos. Entre as causas prováveis para esses percentuais, quais são as mais recorrentes nas organizações?
5. Ferramentas são apoios importantes para o desenvolvimento de software. Por que a camada de ferramentas da Figura 1.1 não funciona adequadamente sem as camadas inferiores?
6. São comuns nas equipes de desenvolvimento de software, profissionais responsáveis por várias funções como analistas, projetistas, programadores simultaneamente em um mesmo projeto. Quais são as implicações dessa organização de equipes?
7. É correto considerar um processo de desenvolvimento de software como um dos processos organizacionais tais como, vendas, marketing, finanças, recursos humanos etc.?
8. A Engenharia de Software é apoiada em diversas tecnologias. Porém, exige-se muita criatividade também para desenvolver um software. A Engenharia de Software pode inibir essa criatividade?
9. De que maneira a Engenharia de Software pode ser efetiva nos projetos de software?
10. Após 40 anos de Engenharia de Software, pode-se dizer que ainda se vive uma crise do software, considerando a falta de atendimento de prazos, custos e de qualidade?

## SUGESTÕES DE LEITURA

BARTEL, Timothy; FINSTER, Mark. A TQM process for systems integration: getting the most from COTS software. *Information System Management*, vol. 12, issue 3, pp. 19-29, Summer, 1995.

EVELEENS, J. Laurenz; VERHOEF, Chris. The rise and fall of the chaos report figures. *IEEE Software*, vol. 27, issue 1, pp. 30-36, Sep., 2010.

NAUR, Peter; RANDELL, Brian. (Eds.). *Software engineering: a report on a conference sponsored by the NATO Science Committee*, Garmisch, Germany, pp. 7-11, Oct., 1968. Brussels: Scientific Affairs Division, NATO, 1969.

PRESSMAN, Roger S. *Software engineering – a practitioner’s approach*. 6<sup>th</sup> edition. New York: McGraw Hill, 2007.

PFLEEGER, Shari L. *Software engineering: theory and practice*. 2<sup>nd</sup> edition. New Jersey: Prentice Hall, 2001.

SHARON, David; ANDERSON, Tracey. A complete software engineering environment. *IEEE Software*, vol.14, issue 2, pp. 123-127, Mar./Apr., 1997.

SOMMERVILLE, Ian. *Engenharia de Software*. Tradução: Kalinka Oliveira e Ivan Bosnic. Revisão Técnica: Kechi Hiramã. 9. ed. São Paulo: Prentice Hall, 2011.

---

### 1.3. IMPORTÂNCIA DOS MÉTODOS DE DESENVOLVIMENTO DE SOFTWARE

*Em um projeto de software existem, em geral, muitos profissionais envolvidos. Têm-se, entre outros, gerentes, analistas, arquitetos, programadores e testadores. Durante a realização das atividades de desenvolvimento, a comunicação entre eles é fundamental. Para estabelecer um canal de comunicação uniforme, é necessário aplicar métodos definidos em processos de desenvolvimento de software. Os métodos atuais são baseados em duas abordagens consolidadas na Engenharia de Software: uma abordagem chamada Estruturada e a outra Orientada a Objetos.*

A importância dos métodos de desenvolvimento de software advém do fato de que eles definem, por meio de suas notações, um canal de comunicação uniforme entre os membros da equipe de desenvolvimento. Também, os métodos estabelecem produtos de trabalho padronizados que facilitam as atividades de manutenção de software. Além disso, permitem que novos colaboradores sejam treinados melhorando a qualidade do software. Os métodos podem ser usados como referência para a escolha e aquisição de ferramentas CASE que podem dar maior produtividade ao processo de desenvolvimento de software.

Tais métodos vêm sendo desenvolvidos desde o início da definição da Engenharia de Software e, atualmente, muitos desses métodos estão consolidados na comunidade de software. Atualmente, duas abordagens são reconhecidas: Estruturada e Orientada a Objetos, que são discutidas na seção a seguir e no Capítulo 4 deste livro. Um trabalho importante de consolidação de conceitos e de boas práticas da Engenharia de Software é o SWEBOK (*Software Engineering Body of Knowledge*).<sup>7</sup> Trata-se de um conjunto de conhecimentos reconhecidos mundialmente. Nesse trabalho, um método estabelece uma notação (ou um conjunto de notações) e provê um processo para guiar a aplicação desse método. Para enfatizar a importância dos métodos de desenvolvimento de software, é necessário definir o que é um processo. Entre as definições encontradas na literatura, um processo é um conjunto de atividades que leva à produção de um produto de software.<sup>8</sup> Um processo define *quais atividades* devem ser realizadas e *um método* para realizá-las. Os processos evoluíram para explorar as capacidades das pessoas em uma organização e as características específicas dos sistemas que estão sendo desenvolvidos. No Capítulo 2 deste livro, são discutidos alguns dos processos mais conhecidos. No Capítulo 11, são discutidos os modelos de maturidade de processos para obter maior qualidade e produtividade de software.

Assim, os métodos também evoluíram ao longo dos anos acompanhando a evolução dos processos de software. Igualmente, as ferramentas também acompanharam essa evolução, apoiando as atividades de desenvolvimento. Existem muitas ferramentas CASE no mercado que apoiam vários métodos conhecidos (veja mais detalhes no Capítulo 9).

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

Quando se vai construir um artefato de software, existem várias maneiras de implementá-lo. Algumas delas são totalmente empíricas e outras usam técnicas reconhecidas para chegar a um bom resultado. Pode-se dizer que um método é uma maneira de padronizar os artefatos que



possam ser compreendidos pelos membros da equipe de desenvolvimento.

Existem também outras denominações para método. Podem ser caracterizados como roteiros e procedimentos. Nas normas ISO (*International Organization for Standardization*) 9001:2000,<sup>9</sup> um método é chamado Instrução de Trabalho. Em uma Instrução de Trabalho, tem-se uma descrição operacional passo a passo de como uma atividade deve ser realizada orientando a elaboração de um artefato ou a realização de um serviço.

Nesta seção destacou-se a importância dos métodos de desenvolvimento de software. A aplicação de métodos torna os produtos uniformes e padronizados e facilita a comunicação entre os membros da equipe. Conforme discutido, cada projeto pode aplicar um método específico. O importante é ter um método que seja efetivamente aplicado. A padronização de documentos faz parte deste método.

## EXERCÍCIOS

1. Existem na literatura muitos métodos de desenvolvimento de software. Você aplica algum desses métodos regularmente aos seus projetos? Se não, quais seriam os motivos?
2. Os métodos burocratizam ou racionalizam as tarefas de desenvolvimento de software? Eles são necessários?
3. Em que medida os métodos prejudicam a criatividade dos desenvolvedores?
4. Os métodos contribuem para a melhoria da qualidade do software? Quais características de software podem ser melhoradas?
5. Uma das consequências de aplicar métodos é a possibilidade de padronizar o desenvolvimento de software. Uma maneira é obter uma documentação uniforme em todos os projetos de software. Qual é a importância de uma documentação uniforme?
6. Podem-se usar métodos sem processos associados? Justifique.
7. Os métodos aplicados em desenvolvimentos de software, na maioria das vezes, não são formalizados nas organizações. Quais são os impactos no software?

8. De que maneira uma ferramenta pode apoiar o desenvolvimento de software? Pode haver incremento de produtividade?
9. Quais são as características de sistemas desenvolvidos pela abordagem Estruturada e Orientação a Objetos?
10. É certo afirmar que as abordagens Estruturada e Orientação a Objetos são aplicáveis a qualquer tipo de software? Quais são as diferenças?

## SUGESTÕES DE LEITURA

BOURQUE, Pierre; DUPUIS, Robert; TRIPP, Leonard L. (Eds.). *Guide to the software engineering body of knowledge – SWEBOK*. USA: IEEE Computer Society Press, 2004.

PFLEEGER, Shari L. *Software engineering: theory and practice*. 2<sup>nd</sup> edition. New Jersey: Prentice Hall, 2001.

PRESSMAN, Roger S. *Software engineering – a practitioner’s approach*. 6<sup>th</sup> edition. New York: McGraw Hill, 2007.

SOMMERVILLE, Ian. *Engenharia de Software*. Tradução: Kalinka Oliveira e Ivan Bosnic. Revisão Técnica: Kechi Hiramã. 9. ed. São Paulo: Prentice Hall, 2011.

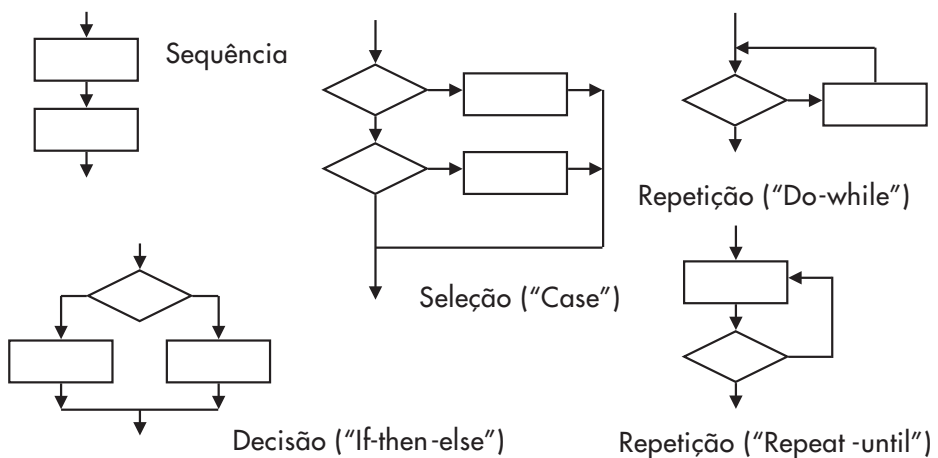
## 1.4. ABORDAGENS ESTRUTURADAS E ORIENTADA A OBJETOS

*Os métodos de desenvolvimento de software têm sido propostos desde a década de 1970. Os métodos foram baseados em duas abordagens: Estruturada e Orientada a Objetos. A primeira abordagem tratou de representar funções do software a partir de construções básicas, desde a programação, passando pelo projeto até a análise estruturada. A segunda abordagem tratou de representar as classes do software, partindo da ideia de representar os objetos do mundo real. Ambas possibilitam desenvolver sistemas em várias áreas de aplicação. Atualmente, os métodos que seguem a abordagem Orientada a Objetos têm tido maior aceitação devido à notação UML (Unified Modeling Language), com o apoio da OMG (Object Management Group), ter se tornado um padrão da indústria para o desenvolvimento de software.*

O termo “abordagem” (em inglês, *approach*) dentro da Engenharia de Software tem um sentido amplo na organização de funções e unidades de um sistema de software que são representados, desenvolvidos e mantidos ao longo de seu ciclo de vida.

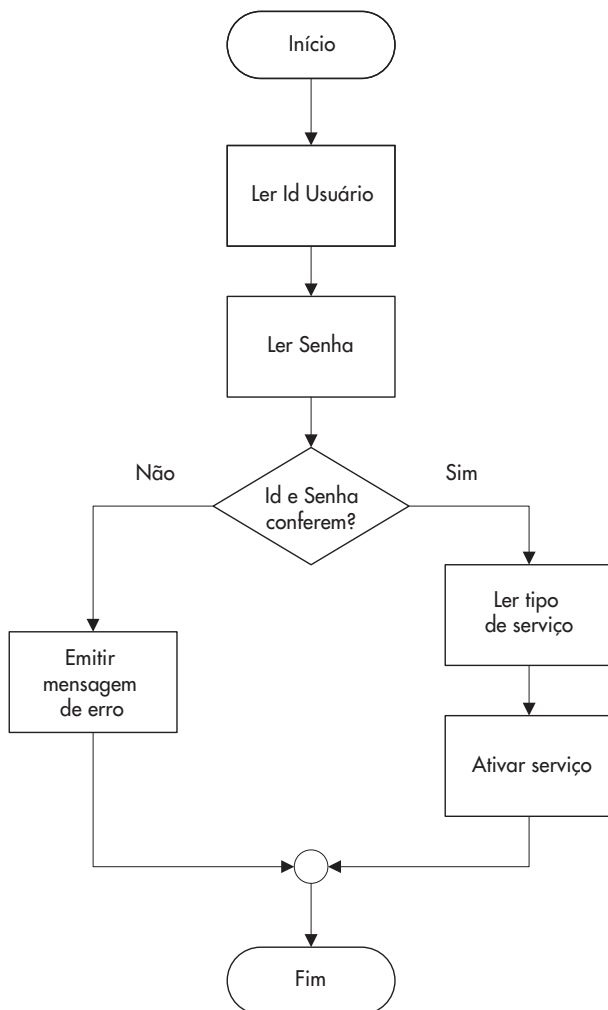
As abordagens Estruturada e Orientada a Objetos são resultados de diversas contribuições realizadas por pesquisadores e desenvolvedores da Engenharia de Software desde a década de 1960.

Os conceitos e as técnicas da abordagem Estruturada originaram-se na década de 1970 da programação estruturada (linguagens Fortran, Algol etc.). A noção “Estruturada” está associada ao uso de construções básicas para a modelagem do fluxo de controle de um software. As construções básicas são sequência, decisão, seleção e repetição, conforme ilustrada na Figura 1.2.



**Figura 1.2** – Construções básicas da linguagem estruturada.

As construções básicas podem ser combinadas e obter o fluxograma de um software, conforme a Figura 1.3. Um “retângulo” de uma construção básica pode ser substituído por quaisquer outras construções básicas formando um fluxo de controle mais complexo.



**Figura 1.3** – Fluxograma de Ativação de Serviço.

O fluxograma é um modelo procedural ou procedimental de um software. Cada elemento do fluxograma representa uma ação ou um passo que o software deve realizar para atingir um resultado ou estado. No caso da Figura 1.3, pode-se ver que são usadas duas construções básicas (sequência e decisão) que foram combinadas para ativar ou não um serviço, baseado no reconhecimento do usuário pelo software. Caso não seja possível, uma mensagem de erro é emitida.

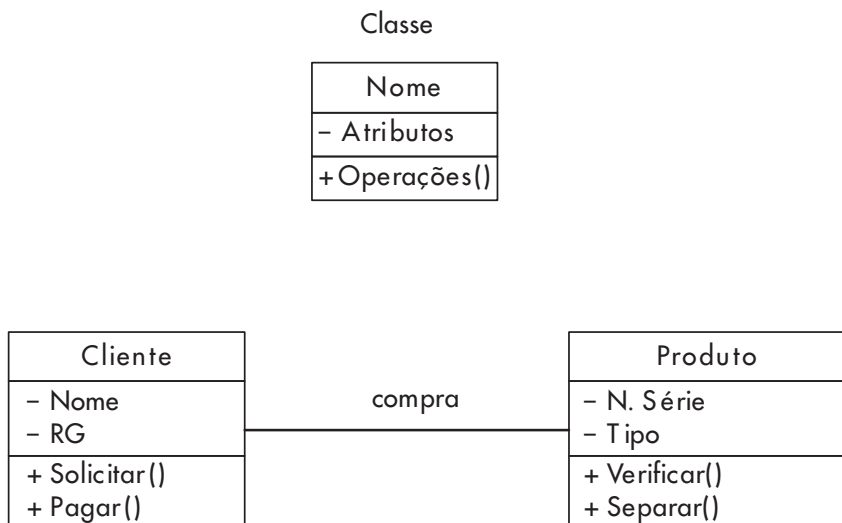
À medida que os programas eram desenvolvidos, alguns pesquisadores buscavam notações gráficas para representar unidades de software mais complexas. O fluxograma era muito limitado para essas representações. Eles estavam preocupados em como tratar dependências entre as unidades de software. A representação encontrada foi denominada arquitetura de software, que representa as unidades inter-relacionadas de software, estabelecendo-se assim, o conceito de projeto estruturado.

Similarmente, os pesquisadores buscavam formas para representar o software em níveis maiores de abstração tentando diminuir a distância entre o problema (necessidades de negócio) e a solução, usando notações mais abstratas introduzindo conceitos de funções ou transformações. Em seguida, estabeleceu-se a análise estruturada (veja mais detalhes na seção 4.2). Neste sentido, a análise estruturada está ligada às necessidades do cliente, o projeto estruturado à solução arquitetural e a programação estruturada à implementação do software, usando uma linguagem estruturada tal como C, Cobol, Fortran etc.

A outra abordagem é conhecida como Orientada a Objetos. Ela tem raízes na linguagem de programação Simula, da década de 1960, considerada a primeira linguagem Orientada a Objetos. Em termos gerais, essa abordagem começou a ter mais visibilidade na década de 1980 quando gradativamente foi sucedendo a Estruturada. A abordagem Orientada a Objetos surgiu da dificuldade da Estruturada em associar elementos de software ao mundo real. Portanto, a passagem do problema para a solução seria mais facilmente realizada, diminuindo-se a distância entre eles. Essa facilidade também acarretaria uma manutenção de software com menor esforço e custo.

Na abordagem Orientada a Objetos, a construção básica é a classe. Uma classe é representada por um retângulo com três compartimentos, sendo um para nome, o outro para atributos e o último para operações da classe. Uma classe é ligada à outra por um relacionamento, representado por uma linha com rótulo. Ele serve para estabelecer a dependência entre duas ou mais classes. Um exemplo pode ser visto na Figura 1.4.

A Figura 1.4 representa um software cuja aplicação poderia ser Compra de Produtos onde os clientes fazem as solicitações e pagam pelos produtos através da classe Cliente e os produtos são verificados e separados através da classe Produto.



**Figura 1.4** – Classe e relacionamento entre as classes Cliente e Produto.

Existem inúmeros métodos que seguem a abordagem Orientada a Objetos. Esses métodos inspiraram o desenvolvimento da notação UML (*Unified Modeling Language*) da OMG (*Object Management Group*) que se tornou um padrão da indústria para o desenvolvimento de software (veja mais detalhes na seção 4.2).

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

As abordagens de representação de software surgiram para administrar a complexidade dos softwares. Para isso, estabelecem regras e notações que possibilitam o entendimento, a modelagem, estruturação e decomposição do software. Detectou-se há muito tempo que somente a codificação não

seria suficiente para dar uma visão abrangente do software cada vez mais complexo.

Outro motivo muito sentido nos projetos de software é a distância entre o que foi solicitado (problema) e o que foi desenvolvido (solução). Essa distância é conhecida como “gap” semântico. As abordagens tentam minimizar esta distância.

Tanto a abordagem Estruturada como a Orientada a Objetos têm pontos fortes e fracos.

A ideia principal da abordagem Estruturada é dar um tratamento sistemático ao software desde o início do seu desenvolvimento. Em essência, na análise estruturada, o software é representado de maneira hierárquica tendo na raiz o sistema de software, subsistemas e funções mais detalhadas nas suas folhas. Pode-se dizer que a análise estruturada dá uma visão top-down do software. Essa representação é muito importante para manter o escopo do software durante o desenvolvimento. Porém, nesta abordagem, a distância semântica não é bem resolvida, pois à medida que se avança nas fases do ciclo de vida, novas representações são criadas no projeto estruturado. E, por fim, na programação estruturada, novamente a representação muda para unidades de software. Estas mudanças de representação, onde erros de interpretação podem ser cometidos, são pontos fracos desta abordagem. Um ponto forte seria a manutenção do escopo do software desde o início do projeto. Outro ponto forte é a visão funcional do software mais intuitiva para os envolvidos no projeto.

No caso da abordagem Orientada a Objetos, ela surgiu com a ideia de associar, desde o início, as entidades ou os objetos do mundo real ao software. À medida que novos objetos são definidos e associados entre si durante o desenvolvimento, o sistema de software torna-se visível. Porém, o tipo de representação do software não se altera. Essa característica é um dos pontos fortes desta abordagem. Outro ponto forte são os objetos do domínio do problema associados diretamente ao software. Essa característica também facilita a manutenção do software e o reuso de objetos.

Um ponto fraco seria a definição tardia do sistema, após muitos objetos terem sido definidos.

Nas duas abordagens, os pontos fracos aumentam ou os pontos fortes diminuem o “gap” semântico.

Entretanto, para ambas, os modelos de software, representações baseadas nessas abordagens, permitem que os membros da equipe de desenvolvimento possam se entender e comunicar as suas ideias de maneira mais eficiente.

Como qualquer modelo, há a necessidade de domínio dos elementos de representação para que, ao serem usados, representem as características essenciais do software. Cabe frisar neste ponto que trabalhar com modelos requer experiência. Exige-se por parte do desenvolvedor que ele seja consistente no uso dos elementos da abordagem (sintaxe) e preciso no significado do software (semântica).

Portanto, um modelo de software para ser eficaz deve preservar a sintaxe da representação e usar a semântica do domínio da aplicação.

As abordagens discutidas são as mais antigas e usadas em desenvolvimento de software. Atualmente uma grande parte dos sistemas que rodam em *mainframes* foi desenvolvida ou é mantida seguindo-se a abordagem Estruturada (por exemplo, sistemas críticos financeiros). Já os sistemas em baixa plataforma estão sendo desenvolvidos majoritariamente na abordagem Orientada a Objetos (por exemplo, aplicações de internet). Essas abordagens não são as únicas, sendo a abordagem Orientada a Aspectos uma nova abordagem para lidar com requisitos não funcionais do software (veja mais detalhes na seção 10.3).

## EXERCÍCIOS

1. Existem diferenças entre as duas abordagens discutidas. Cite duas diferenças relevantes de cada abordagem.
2. Em sua opinião, qual das duas abordagens representa melhor um software durante o seu ciclo de vida? Justifique.



3. As abordagens de representação do software se propõem a diminuir o “gap” semântico entre o problema (negócio) e a solução (software). Qual das abordagens discutidas atende melhor a esta expectativa?
4. Quais características de software não são bem representadas pelas abordagens discutidas?
5. Por que definir o software a partir dos objetos do mundo real não é trivial?
6. As operações de uma classe representam as suas responsabilidades no sistema de software. Como se pode verificar que o conjunto das operações de classes representa a funcionalidade especificada do software?
7. As duas abordagens discutidas foram usadas com sucesso em vários projetos de software. Porém, muitos fracassaram por motivos de mudança de prioridades, não atendimento de custos e prazos e questões diversas. Aponte possíveis questões que podem conduzir um projeto ao fracasso nas duas abordagens.
8. Foi discutido que a definição e a manutenção do escopo do software são essenciais para um projeto. Por este critério, a abordagem Estruturada seria a escolhida. Sugira uma forma alternativa para que a abordagem Orientada a Objetos também satisfaça a este critério.
9. A abordagem Estruturada tem pontos fortes e fracos. Você concorda com os pontos discutidos? Por quê?
10. A abordagem Orientada a Objetos tem pontos fortes e fracos. Você concorda com os pontos discutidos? Por quê?

## SUGESTÕES DE LEITURA

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. *The unified modeling language user guide*. 2<sup>nd</sup> edition. USA: Addison-Wesley Professional, 2005.

DeMARCO, Tom. *Análise Estruturada e Especificação de Sistemas*. Rio de Janeiro: Campus, 1989.

GANE, Chris; SARSON, Trish. *Análise estruturada de sistemas*. Rio de Janeiro: LTC, 1983.

JACOBSON, Ivar; CHRISTERSON, Magnus; JONSSON, Patrik; ÖVERGAARD, Gunnar. *Object-oriented software engineering: a use case driven approach*. USA: Addison-Wesley Professional, 1992.

PFLEEGER, Shari L. *Software engineering: theory and practice*. 2<sup>nd</sup> edition. New Jersey: Prentice Hall, 2001.

RUMBAUGH, James; BLAHA, Michael; PREMERLANI, William; EDDY, Frederick; LORENSEN, William. *Object-oriented modeling and design*. Englewood Cliffs: Prentice-Hall, 1991.

YOURDON, Edward. *Análise estruturada moderna*. Rio de Janeiro: Campus, 1990.

## Processos de software

---

### 2.1. CASCATA

---

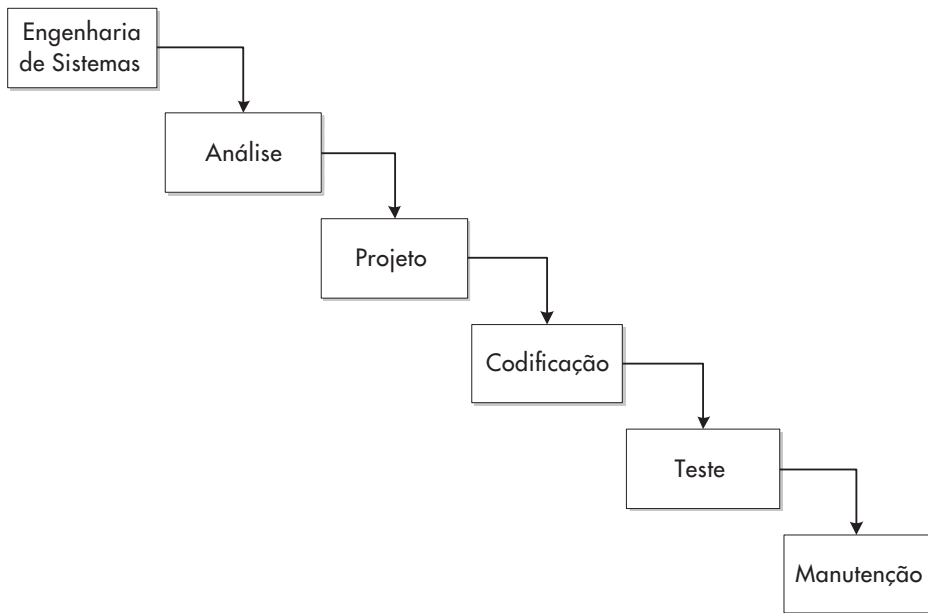
*Processos de software são importantes, pois estabelecem para os membros da equipe de projeto uma diretriz de o que deve ser feito para atender aos objetivos do software. Ou seja, os processos definem quais são as atividades que permitem obter o produto de software. O processo Cascata (em inglês Waterfall Model), proposto na década de 1970, tem um conjunto de fases, em sequência, nas quais o trabalho anterior deve estar finalizado, verificado e aprovado antes de se iniciar a próxima fase. O processo Cascata reflete o que é mais usado em outros projetos de engenharia e é ainda muito usado no desenvolvimento de software, particularmente quando ele faz parte de um projeto maior de Engenharia de Sistemas.*

O processo Cascata, sendo também conhecido como Waterfall Model, é constituído de um conjunto de atividades (Engenharia de Sistemas, Análise, Projeto etc.), conforme ilustrado na Figura 2.1. As setas sobre as atividades indicam a sequência a ser seguida.

As principais características do processo Cascata são:

- a) As atividades de especificação, codificação e testes seguem uma disciplina.

O desenvolvedor deve pensar em uma especificação, ou seja, uma definição do software, antes da codificação que é a implementação daquilo que foi especificado. Após isto, pode-se testar o software que foi implementado.



**Figura 2.1** – Processo Cascata.

Muitas vezes, os desenvolvedores pensam em adiantar a codificação por considerar o software simples ou por estar pressionado pelo prazo do projeto.

- b) Uma atividade não é iniciada sem que a anterior tenha sido encerrada e aprovada.

Cada atividade precisa ser benfeita e o resultado verificado antes de prosseguir para a próxima atividade. Isso garante aos responsáveis pela próxima atividade produtos confiáveis para realizar as suas atividades.

- c) Há uma sequência rígida de atividades.

A sequência lógica de desenvolvimento de software precisa ser respeitada. A rigidez é um alerta aos desenvolvedores para respeitarem o propósito de cada atividade.

d) O usuário/cliente é envolvido somente no início e no fim do processo.

Um importante aspecto de qualidade que é a participação do usuário/cliente no processo não está sendo observado. O usuário/cliente provê as necessidades de negócio e o desenvolvedor trata de materializá-las na forma de software. Se o usuário/cliente é envolvido novamente após o software estar concluído, pode ser tarde e muito dispendioso, se houver necessidade de alguma mudança.

A seguir, são descritas as atividades do processo Cascata.

A atividade de Engenharia de Sistemas tem por objetivo entender as necessidades de negócio do cliente e estabelecer requisitos do ponto de vista sistêmico (hardware, software, banco de dados e pessoas) até definir, em alto nível, o que será desenvolvido em software. O documento típico dessa atividade é uma Especificação de Sistema.

A atividade Análise tem por objetivo entender os requisitos do sistema e detalhar os requisitos do software. Nessa atividade deve-se compreender o domínio da informação, a funcionalidade requerida, o desempenho e as interfaces exigidas. O documento típico dessa atividade é uma Especificação de Requisitos de Software.

Na atividade de Projeto procura-se definir a estrutura de dados, a arquitetura do software, detalhes procedimentais e as interfaces. O documento típico dessa atividade é uma Especificação de Projeto.

A atividade de Codificação tem por objetivo traduzir as especificações de software em códigos que sejam compreendidos por um sistema computacional. Naturalmente, esses códigos são dependentes da linguagem de programação escolhida. O documento típico dessa atividade é a Listagem de Programas.

Na atividade de Teste busca-se verificar os aspectos estruturais e lógicos do software, bem como os seus aspectos sistêmicos, com o intuito descobrir defeitos no software. O documento típico dessa atividade é o Relatório de Testes.

Na atividade de Manutenção pretende-se realizar mudanças no software devido a defeitos encontrados durante a sua operação, novas necessidades de negócio, atualização de plataforma computacional e necessidade de mudança preventiva. O documento típico dessa atividade é o Relatório de Manutenção.

O processo Cascata foi muito usado nas décadas de 1970 e 1980 em projetos de grande porte. No entanto, dadas as demandas cada vez mais exigentes por qualidade e produtividade, esse processo e seus similares foram questionados na década de 1990, com o advento dos métodos ágeis (veja mais detalhes na seção 2.5).

Uma das críticas recorrentes é que este processo não é realista, dado que o levantamento de requisitos não é completo em muitos projetos e que a participação do usuário/cliente fica limitada ao início e ao fim do processo. Os requisitos são revistos pelo usuário/cliente e se alteram durante o desenvolvimento do software. Os requisitos levantados no início não são os mesmos no término do desenvolvimento.

Entretanto, o processo Cascata tem como pontos fortes a produção de documentação intermediária em cada fase e a sua aderência a outros modelos de processo de engenharia. Um ponto fraco é a acomodação de mudanças depois que o processo está em andamento. Os compromissos devem ser assumidos no início do processo, tornando difícil reagir às mudanças de requisitos do cliente.

No entanto, o processo Cascata reflete o que é mais usado em outros projetos de engenharia e é ainda muito usado no desenvolvimento de software, particularmente quando ele faz parte de um projeto maior de Engenharia de Sistemas.

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

Na literatura é comum referir-se ao processo de software como ciclo de vida de software. Conforme pode ser visto no Glossário, *processo* está ligado a um conjunto de atividades relacionadas para atingir um objetivo e *ciclo de vida* a um período entre a concepção e desativação do software. Neste livro, por ser mais geral, adotou-se o termo *processo*. O processo Cascata é tratado como um modelo de processo de software.

Quando um projeto de software inicia, é natural se apoiar em uma estratégia que direcione o que deve ser feito. Isso se torna fundamental para obter uma boa produtividade da equipe. Dizem que todos os caminhos levam à Roma. Da mesma forma, pode-se dizer que todos os processos de software existentes conduzem a uma solução de software, não necessariamente única, porém que atenda às expectativas do cliente.

O que difere um processo de software de outro são as atividades definidas e a sua dinâmica para chegar a uma solução de software. Portanto, cabe aos responsáveis pelo projeto definir qual é o processo a ser seguido. A cada novo projeto, um processo deve ser definido, baseado no tipo de aplicação, prazos, custos, recursos e riscos.

Nesta seção destacou-se o processo Cascata, suas características e suas particularidades. Sendo um modelo clássico, a maioria dos sistemas legados foi desenvolvida seguindo-se este processo. Apesar da sua inadequação nos projetos atuais, é um processo importante que aponta para a necessidade de uma especificação benfeita antes da codificação e dos testes de software. Esta é a grande contribuição do processo Cascata para os outros processos que surgiram posteriormente.

## EXERCÍCIOS

1. Uma das vantagens do processo Cascata é disciplinar as atividades de desenvolvimento de software. Você concorda?
2. Uma das desvantagens é a imposição de uma sequência rígida de atividades. Quais são os impactos desta imposição para o desenvolvimento de software?
3. Uma das críticas ao processo Cascata é que os projetos de software seriam mais caros. Você concorda? Justifique.
4. Se ao fim de cada fase os resultados devem ser verificados e aprovados no processo Cascata, quais são os impactos para o desenvolvimento de software?
5. De que maneira o processo Cascata pode ser flexibilizado para aumentar seus pontos fortes e diminuir seus pontos fracos, tornando-o menos rígido?

6. Como os requisitos podem ser considerados completos no processo Cascata?
7. O processo Cascata contribui para a manutenibilidade do software. Você concorda? Justifique.
8. O teste é uma atividade importante. Porém, no processo Cascata, ela é a última atividade de desenvolvimento. Se for descoberto um defeito, que implique em mudanças nas especificações geradas nas fases iniciais do processo, como os impactos ao projeto podem ser minimizados?
9. Se uma mudança no software em desenvolvimento for solicitada em fases adiantadas do processo Cascata, o custo dessa mudança é muito alto. De que maneira os custos podem ser reduzidos?
10. Por que aceitar mudanças de requisitos do cliente é tão difícil no processo Cascata?

## SUGESTÕES DE LEITURA

PFLEEGER, Shari L. *Software engineering: theory and practice*. 2<sup>nd</sup> edition. New Jersey: Prentice Hall, 2001.

PRESSMAN, Roger S. *Software engineering – A practitioner's approach*. 6<sup>th</sup> edition. New York: McGraw Hill, 2007.

ROYCE, Winston W. Managing the development of large software systems. In: *Proceedings, IEEE WESCON*, pp. 1-9, Aug., 1970.

SOMMERVILLE, Ian. *Engenharia de Software*. Tradução: Kalinka Oliveira e Ivan Bosnic. Revisão Técnica: Kechi Hiramã. 9. ed. São Paulo: Prentice Hall, 2011.



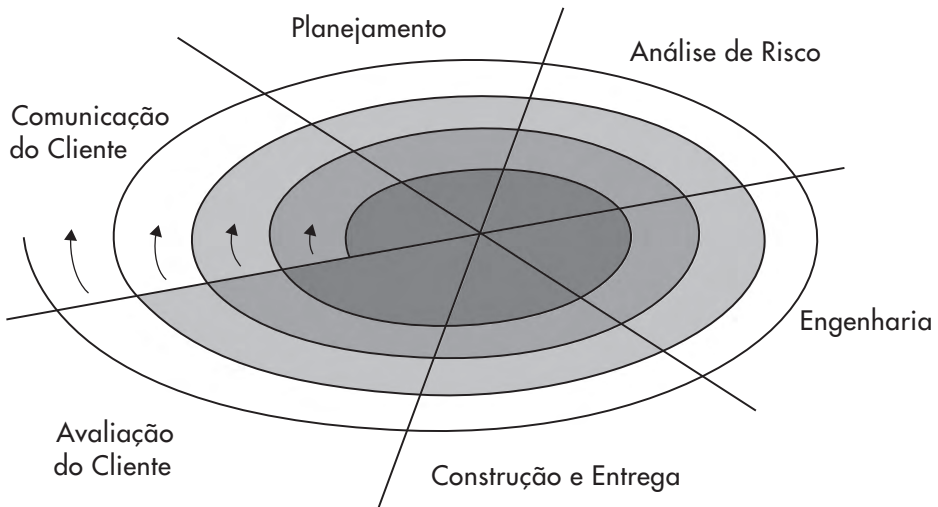
## 2.2. EVOLUTIVO

*Para minimizar os pontos fracos do processo Cascata, surgiram os processos do tipo Evolutivo, que permitem que se trabalhe com um subconjunto de requisitos do produto completo os quais são incrementados, gradualmente, e entregues aos clientes. Neste processo, a complexidade do software pode ser mais bem administrada, minimizando-se os riscos. No entanto, um dos pontos fracos do processo Evolutivo é saber se essa característica evolucionária é controlável. O fato de cada incremento poder ser avaliado pelo cliente melhora a qualidade final do software. Porém, ainda é necessário saber se os processos Evolutivos são realmente eficazes.*

Os processos Evolutivos surgiram para atender aos requisitos de negócio e de produto que se alteravam durante o desenvolvimento do software e os prazos de mercado que não eram compatíveis com um produto de software completo. Neste cenário, o lançamento de uma versão limitada necessária para atender às pressões de negócio e à competitividade de mercado passou a ser mais urgente. Para isso, os processos Evolutivos iniciam com um subconjunto de requisitos do produto ou sistema que é desenvolvido e incrementado gradualmente. Cada incremento é entregue ao cliente.

O processo Cascata define um desenvolvimento linear e sequencial do software. Da maneira como é proposto, ele não se adapta facilmente às novas exigências de mercado. Enquanto o processo Cascata é focado na entrega de um sistema completo no fim do seu processo, os processos Evolutivos são iterativos: o software é desenvolvido evolutivamente em direção ao produto final cada vez mais completo.

A Figura 2.2 apresenta um bom representante de processo Evolutivo conhecido como Espiral, que é uma variação do modelo proposto por Barry Boehm,<sup>9</sup> em 1988. As atividades do processo Espiral evoluem do centro para fora, no sentido horário.



**Figura 2.2** – Processo Espiral.<sup>3</sup>

As principais características do processo Espiral são:

- a) O planejamento associado com a engenharia, ou seja, disciplinas de gerenciamento do projeto estão integradas com as de engenharia. Baseado no levantamento de requisitos com o cliente, um plano de desenvolvimento deve ser feito. Diferentemente do processo Cascata, o processo Espiral explicita atividades gerenciais junto com a engenharia.
- b) A disciplina de análise de risco diminui os riscos de desenvolvimento do software. Antes de iniciar o desenvolvimento propriamente dito, uma análise de risco permite verificar se ele é viável técnica e gerencialmente.
- c) Há o envolvimento do cliente em cada iteração de maneira ativa no desenvolvimento do software. O cliente é sempre envolvido a cada iteração ou produto obtido. Nesse momento, o cliente pode avaliar o produto e essa avaliação pode ser considerada na próxima iteração.

O processo Espiral combina o modo iterativo de prototipação (geração de protótipos) com o modo controlado e sistemático dos processos sequenciais lineares (por exemplo, o processo Cascata), favorecendo o desenvolvimento de versões incrementais do software. Neste processo, o software é

desenvolvido em uma série de entregas incrementais. Durante as iterações iniciais, uma entrega incremental pode ser um modelo em papel ou protótipo. Nas iterações mais avançadas, versões cada vez mais completas do sistema são produzidas.

O processo Espiral é dividido em uma série de atividades, chamadas regiões de tarefas. Tipicamente, existem entre 3 e 6 regiões. No caso da Figura 2.2, o processo Espiral contém 6 regiões, nas quais são realizadas as seguintes atividades:<sup>3</sup>

- Comunicação do Cliente, que visa a estabelecer uma comunicação efetiva entre desenvolvedor e cliente.
- Planejamento, em que se definem recursos, cronograma e outras informações de projeto.
- Análise de Risco, na qual se avaliam os riscos técnicos e gerenciais.
- Engenharia, que consiste na implementação de uma ou mais representações do software.
- Construção e Entrega, quando se implementa, testa, instala o software e se provê apoio ao usuário.
- Avaliação do Cliente, que objetiva obter realimentação do cliente baseada na avaliação das representações do software implementadas durante as atividades de engenharia e construção e entrega.

O processo Espiral se move a partir do seu centro no sentido horário. Cada rodada representa um tipo de projeto. Por exemplo, projeto de concepção, projeto de desenvolvimento, projeto de melhoria e projeto de manutenção de software. Assim, pode-se adaptar o processo Espiral ao processo completo de um software.

O processo Espiral é uma abordagem mais realística para o desenvolvimento de sistemas e softwares de grande porte. Devido à sua característica progressiva, o desenvolvedor e o cliente compreendem e reagem melhor aos riscos em cada nível evolucionário. Diferentemente do processo Cascata, o processo Espiral demanda uma atenção direta sobre os riscos técnicos e gerenciais em todas as etapas do projeto que, se adequadamente aplicada, deve reduzir os riscos antes que eles se tornem problemáticos.

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

Uma realidade nos projetos de software é que os requisitos iniciais, invariavelmente, sofrem mudanças durante o projeto. Isso envolve manter a consistência do software após a incorporação de mudanças. Assim, a estrutura dos processos pode facilitar ou não as mudanças. Os processos Evolutivos permitem que mudanças sejam mais facilmente incorporadas ao software durante o projeto, diferentemente do processo Cascata. A comunicação regular do cliente permite que as mudanças sejam consideradas durante as atividades de planejamento e análise de riscos.

Embora protegido pelos ciclos sucessivos em direção ao produto final, o escopo do software pode ser bastante alterado. Não há clareza sobre a manutenção dos prazos, custos e recursos estimados originalmente para o projeto.

Uma das dificuldades do processo Espiral é saber se essa característica evolucionária é controlável, ou seja, se o número de iterações para desenvolver um software pode ser controlado e previsto. Mas, o fato de cada incremento poder ser avaliado pelo cliente melhora a qualidade final do software.

Esta dinâmica pode ser encontrada em outros modelos de processos Evolutivos conhecidos como Incremental e Prototipação (ver mais detalhes nas leituras sugeridas). Ainda é necessário saber se os processos Evolutivos são realmente eficazes.

## EXERCÍCIOS

1. Como o número de iterações pode ser estimado no processo Espiral?
2. Dadas as necessidades do cliente, como os requisitos devem ser definidos para serem implementados em iterações sucessivas?
3. Cite um risco técnico e outro gerencial passíveis de serem analisados na atividade de análise de risco.
4. Em que região do processo Espiral os requisitos do usuário podem ser levantados?

5. O processo Espiral exige mais atividades de gerenciamento do que o processo Cascata? Justifique.
6. De que maneira um ciclo completo como um processo Cascata poderia ser incorporado ao processo Espiral?
7. Em quais aspectos o processo Espiral é mais eficaz que o processo Cascata?
8. Qual seria o perfil adequado do cliente para apoiar o processo Espiral?
9. Qual seria um critério para determinar o fim de um projeto em um processo Espiral que pode ter requisitos iniciais alterados?
10. O processo Espiral apresenta vantagens e desvantagens em relação ao processo Cascata. Cite duas vantagens e desvantagens do processo Espiral.

### SUGESTÕES DE LEITURA

BOEHM, Barry W. A spiral model of software development and enhancement. *IEEE Computer*, vol. 21, issue 5, pp. 61-72, May, 1998.

BOEHM, Barry W. Anchoring the software process. *IEEE Software*, vol. 13, issue 4, pp. 73-82, Jul., 1996.

PFLEEGER, Shari L. *Software engineering: theory and practice*. 2<sup>nd</sup> edition. New Jersey: Prentice Hall, 2001.

PRESSMAN, Roger. S. *Software engineering – A practitioner's approach*. 6<sup>th</sup> edition. New York: McGraw Hill, 2007.

SOMMERVILLE, Ian. *Engenharia de Software*. Tradução: Kalinka Oliveira e Ivan Bosnic. Revisão Técnica: Kechi Hiramã. 9. ed. São Paulo: Prentice Hall, 2011.

## 2.3. MODELO V

O Modelo V é similar ao processo Cascata, com a diferença de que este modelo está preocupado com o planejamento dos testes nas fases de desenvolvimento. Cada tipo de teste é realizado seguindo um plano de testes correspondente, verificando e validando se as especificações definidas nas fases de desenvolvimento estão sendo atendidas. Seguindo a característica sequencial do processo Cascata, um teste de mais alto nível é realizado somente se o teste atual for bem-sucedido. De forma geral, o Modelo V não é tão conhecido como o processo Cascata, mas pela sua simplicidade e melhor organização das atividades de teste ele se torna um processo bastante interessante para disseminar a importância dos testes no processo de desenvolvimento de software.

O Modelo V é um processo para o desenvolvimento de sistemas, inspirado no processo Cascata e usado como padrão em projetos de software na Alemanha. No Modelo V, o lado esquerdo dá ênfase à decomposição de requisitos e o lado direito à integração do sistema, conforme pode ser visto na Figura 2.3.

Em linhas cheias, é representado o fluxo de atividades de desenvolvimento e, em linhas tracejadas, o fluxo de verificação e validação, destacando-se os testes.

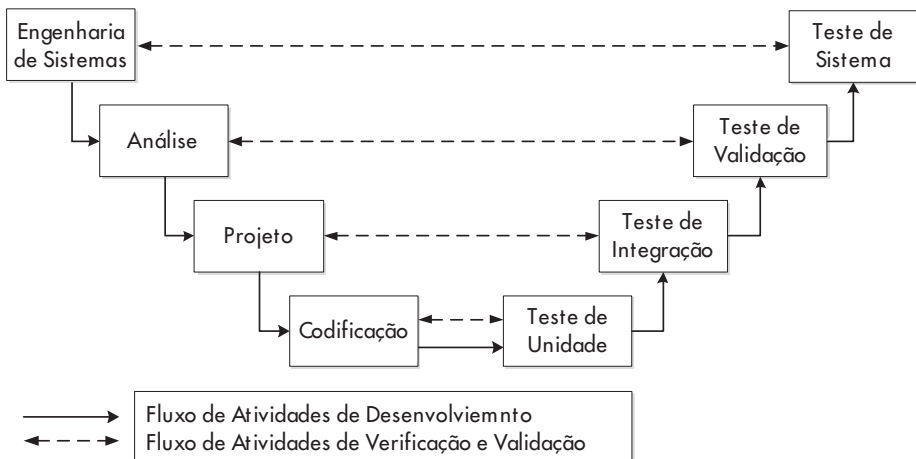


Figura 2.3 – Modelo V.

As principais características do Modelo V são:

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

Em relação ao processo Cascata, as fases do lado esquerdo têm os mesmos objetivos (descritos na Seção 2.1). As fases do lado direito destacam os tipos de teste que devem ser realizados para verificar e validar se as especificações estão sendo atendidas.

Todos os testes realizados do lado direito têm a especificação correspondente a ser verificada do lado esquerdo do Modelo V. O teste de unidade deve ser realizado verificando se as especificações de módulos de software usadas na fase de codificação estão sendo atendidas. O teste de integração deve ser realizado verificando se a especificação de projeto, representada pela arquitetura do software da fase de projeto, está sendo atendida. O teste de validação deve ser realizado verificando se as expectativas do cliente descritas na especificação de requisitos de software da fase de análise estão sendo atendidas. O teste de sistema deve ser realizado verificando se a especificação de sistema (hardware, software, banco de dados, pessoas) está sendo atendida. Esse último é realizado normalmente no ambiente do usuário.

Uma vantagem do Modelo V é que, se alguns problemas forem detectados durante a verificação e validação, as especificações correspondentes devem ser corrigidas antes de se realizar os testes de nível mais alto. O Modelo V deixa os objetivos dos testes mais explícitos do que o processo Cascata. Uma característica do Modelo V é que ele é focado nas atividades de desenvolvimento e correção, enquanto o processo Cascata é focado nos documentos e artefatos.

### REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

Independentemente do processo, das fases e das atividades definidas, sempre haverá a necessidade de realização de testes em maior ou menor grau. De fato, há um sentimento geral de que um software sem teste não é confiável e pode trazer prejuízos. Logo, um software deve ser testado antes de entrar em produção.

O grande dilema é se os testes foram previstos no projeto com prazos e recursos adequados. Porém, constata-se que os projetos de software invariavelmente estão atrasados e com isso os testes acabam sendo comprometidos ou são realizados com menos rigor.

O Modelo V não resolve os problemas mencionados, porém exige dos responsáveis pelo menos a reflexão sobre uma disciplina coerente de testes. Os testes não correspondem meramente à imposição de entradas e à obtenção de resultados de um software. Tem de haver uma estratégia para que o software seja testado, incluindo suas partes integrantes. O encaminhamento gradual dos testes a partir do módulo até o sistema permite obter um software com mais qualidade.

O Modelo V é um processo complementar ao processo Cascata, pois preserva muitas de suas características. Significa também que as vantagens e desvantagens do processo Cascata persistem. No entanto, o teste é uma atividade fundamental para verificar se o que foi desenvolvido atende à sua especificação. A identificação dos testes em níveis de granularidade ajuda a disciplinar as atividades de teste nos projetos de software.



## EXERCÍCIOS

1. Um software desenvolvido deve ser testado. Todos os processos de software conhecidos preveem testes. O Modelo V explicita diversos tipos de testes. Qual é a importância deste processo?
2. Em sua opinião, quando os testes devem ser planejados e especificados?
3. O conceito de “teste integrado” muitas vezes é confundida com “teste de validação”. Qual é a diferença entre esses testes?
4. Na verificação, estamos “verificando” se o software está de acordo com as especificações. Na validação, estamos “verificando” se o software está de acordo com as expectativas do cliente. Dê um exemplo de verificação e validação de software.
5. O Modelo V é uma extensão do processo Cascata. Em que sentido o Modelo V é melhor do que o processo Cascata?
6. Pode-se suprimir o teste de integração, mantendo-se o de unidade e o de validação? Quais são as implicações?
7. O Modelo V pode ser adaptado para outros tipos de testes. Dê um exemplo de teste que possa ser incluído no Modelo V. Explique o que esse teste estaria verificando.
8. Deve-se escolher um processo de software para cada tipo de projeto. Dê exemplos de aplicações onde o Modelo V seja mais aplicável.
9. O Modelo V pode ser aplicado em projetos que seguem as abordagens Estruturada ou Orientada a Objetos. Você concorda? Justifique.
10. O Modelo V pode ser adaptado para um processo Evolutivo? De que maneira ele poderia trabalhar com iterações?

## SUGESTÕES DE LEITURA

PFLIEGER, Shari L. *Software engineering: theory and practice*. 2<sup>nd</sup> edition. New Jersey: Prentice Hall, 2001.

PRESSMAN, Roger S. *Software engineering – a practitioner’s approach*. 6<sup>th</sup> edition. New York: McGraw Hill, 2007.

SOMMERVILLE, Ian. *Engenharia de Software*. Tradução: Kalinka Oliveira e Ivan Bosnic. Revisão Técnica: Kechi Hirama. 9. ed. São Paulo: Prentice Hall, 2011.

## 2.4. PROCESSO UNIFICADO

Diferentemente dos processos anteriores, o Processo Unificado (UP – Unified Process) destaca as fases que correspondem à parte dinâmica do processo e os fluxos de trabalho que correspondem às disciplinas do desenvolvimento de software. Pode-se dizer que o UP é um modelo híbrido de processo que reúne as perspectivas dinâmicas (fases) e estáticas (fluxos de trabalho) em um único processo. O UP foi proposto no cenário da abordagem Orientada a Objetos. Em particular, o processo é dirigido a Casos de Uso (uma técnica para especificar uma função do software do ponto de vista do usuário). Em seguida, uma estratégia de projeto é definida, por exemplo, a partir dos Casos de Uso mais críticos. Uma arquitetura do software é então elaborada até chegar a um entregável (release). Um dos processos derivados do UP é o RUP (Rational Unified Process), da empresa IBM/Rational.

O Processo Unificado foi uma proposta desenvolvida para unificar uma série de abordagens existentes e servir como guia para os desenvolvedores de software. A Figura 2.4 ilustra esse processo.

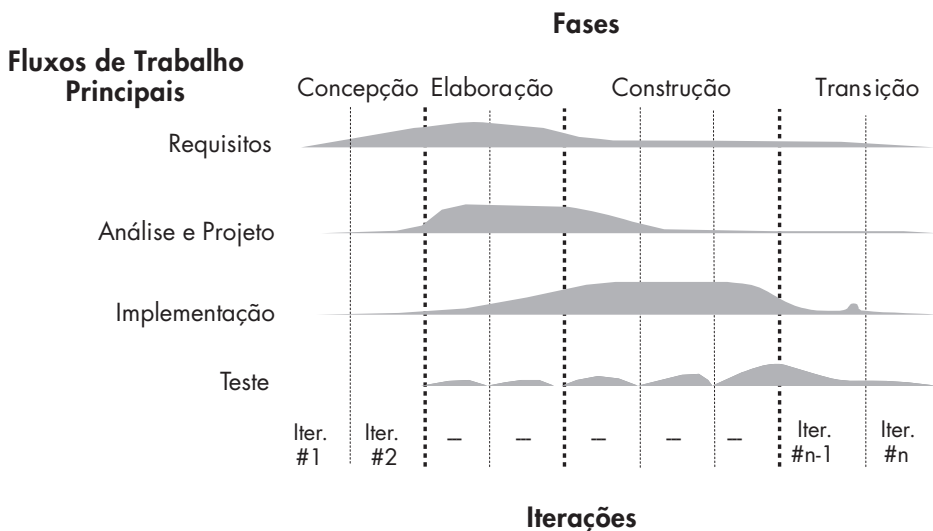


Figura 2.4 – Processo Unificado (UP).<sup>10</sup>

As suas principais características do processo UP são:<sup>10</sup>

- a) É estruturado em duas dimensões: fases e fluxos de trabalho (*workflows*).  
Diferentemente dos processos anteriores, o UP destaca as fases que correspondem à parte dinâmica e os fluxos de trabalho que correspondem às disciplinas do desenvolvimento de software. Pode-se dizer que o UP é um modelo híbrido de processo que reúne as perspectivas dinâmicas (fases) e estáticas (fluxos de trabalho) em um único processo.
- b) É dirigido pelos Casos de Uso.  
Caso de Uso é uma técnica para especificar uma função do software do ponto de vista de um ator (usuário ou outro sistema).
- c) É centrado na arquitetura.  
O foco principal é a arquitetura do sistema desde o início. Uma arquitetura apresenta uma visão dos elementos estruturais e comportamentais do sistema.
- d) O produto é desenvolvido de forma iterativa (iterações) e incremental.  
O software é desenvolvido por meio de pequenas versões que vão sendo incrementadas a cada nova iteração até chegar ao sistema completo.

As fases são as seguintes:<sup>10</sup>

- a) Concepção: Nesta fase, definem-se o que é o sistema, a visão preliminar de sua arquitetura e os subsistemas (em alguns casos). Um plano também é elaborado considerando-se os riscos e custos do projeto. Também se desenvolvem os casos de uso mais críticos.
- b) Elaboração: Nesta fase, os casos de uso são especificados em detalhes e uma arquitetura é expressa como visões (modelos estáticos e dinâmicos) do sistema, que, em conjunto, representam o sistema completo. Também se desenvolve um plano detalhado com atividades e recursos requeridos para realizar o projeto.
- c) Construção: Nesta fase, o produto é construído de acordo com a arquitetura em direção ao produto pronto para ser transferido para os usuários. Ao fim desta fase, o produto contém todos os casos de uso acordados entre o desenvolvedor e o cliente.
- d) Transição: Nesta fase, o produto torna-se um entregável (*release*) beta. Denomina-se beta pois o entregável é usado por um número restrito de usuários, que irá reportar eventuais defeitos, deficiências e sugestões de melhoria. Esta fase envolve treinamento, apoio ao usuário e correção de defeitos e incorporação de melhorias, gerando um entregável final para um número maior de usuários.

Os fluxos de trabalho correspondem às disciplinas de Requisitos (levantamento e análise de requisitos), Análise e Projeto (detalhamento dos requisitos e da arquitetura), Implementação (codificação) e Testes.

Uma iteração corresponde a um conjunto de atividades realizadas, segundo os fluxos de trabalho, de acordo com um plano de iterações. Cada iteração resulta em um entregável (*release*).

O processo UP, embora não seja exclusivo, foi desenvolvido no cenário da abordagem Orientada a Objetos, com a proposta da *Unified Modeling Language* (UML). A UML é uma linguagem visual resultante de numerosos métodos orientados a objetos que existiam no início da década de 1990.

Um dos exemplos derivados do trabalho sobre UP e UML é o processo *Rational Unified Process* (RUP) da empresa IBM/Rational. Uma das vantagens do UP/RUP em relação aos processos apresentados nas seções anteriores é a desvinculação das fases com os fluxos de trabalho que podem ser ativados em todos os estágios do processo. Outra vantagem é o reconhecimento de que implantação de software no ambiente do usuário é parte do processo. No entanto, o UP/RUP não é adequado para todos os tipos de desenvolvimento de software, por exemplo, softwares que essencialmente processam uma grande quantidade de dados, como na área de meteorologia, transportes, comunicações etc.

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

Quando se trata de um processo de software, duas características são observadas. A primeira diz respeito às atividades que devem ser realizadas. A segunda diz respeito a sua dinâmica, ou seja, em que ordem as atividades devem ser realizadas. Nos processos de software vistos até aqui, estas características não são bem distinguíveis. Reconhecê-las é importante. As atividades podem ser únicas, porém a forma como elas são interligadas pode ser diferente para cada projeto.

Nesse aspecto, o UP (ou RUP) é bem organizado. É comum também interpretar as atividades como miniprocessos Cascata. Lembrando que o processo UP é iterativo e incremental, o resultado de uma iteração obtido por um processo Cascata é um incremento para a próxima iteração.

O UP tem uma estrutura de processo que não se assemelha aos anteriores. Porém, ele reúne algumas características encontradas nos outros processos. Pode-se dizer que ele é um modelo híbrido de processo. Ou seja, pode ser visto na perspectiva dinâmica (fases) e na perspectiva estática (atividades representadas pelos fluxos de trabalho). Por exemplo, no processo Cascata essas duas perspectivas se confundem, são vistas de forma única. O RUP é mais realista, pois incorporou outros fluxos de trabalho, tais como, modelagem de negócios, implantação, gerenciamento de projetos etc.

## EXERCÍCIOS

1. O processo UP é centrado na arquitetura de software. O que você entende por arquitetura de software?
2. Os casos de uso mais críticos são escolhidos para serem implementados nas primeiras iterações. Cite alguns critérios para definir estes casos de uso.
3. Quando e de que maneira as iterações devem ser integradas para implementar o sistema completo?
4. Como se poderia quantificar o número necessário de iterações para completar um sistema?
5. Embora possa ser considerado um processo híbrido, o processo UP tem mais características de processo Evolutivo. Quais são estas características?
6. O processo UP pode ser aplicado em diversos tipos de software. Cite duas aplicações em que o processo UP não seja adequado.
7. Se um projeto de software já possui uma especificação de requisitos de software, não apoiada em casos de uso, como o processo UP poderia ser usado?
8. O processo UP foi concebido no cenário da abordagem Orientada a Objetos. Ele pode ser usado para a abordagem estruturada? Justifique.

9. Imagine que você precisa escolher um processo para o desenvolvimento de software. Cite três razões para a escolha (ou não) do processo UP.
10. As áreas preenchidas mostradas na Figura 2.4 em cada disciplina representam o esforço necessário em cada fase do processo UP. Por que as áreas não estão distribuídas igualmente em todas as fases?

### SUGESTÕES DE LEITURA

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. *The unified modeling language – user guide*. 2<sup>nd</sup> edition. USA: Addison-Wesley Professional, 2005.

JACOBSON, Ivar; BOOCH, Grady; RUMBAUGH, James. *The unified software development process*. USA: Addison-Wesley Professional, 1999.

## 2.5. EXTREME PROGRAMMING

O processo Extreme Programming (XP) surgiu como uma alternativa para os processos ditos pesados (tradicionais), que não se adaptam rapidamente às mudanças de negócio. Os processos ditos pesados são baseados em um cuidadoso planejamento do projeto e são controlados por um rigoroso processo de desenvolvimento de software. Os métodos ágeis surgiram neste contexto para minimizar os efeitos de sobrecarga (overhead) causados pela rigidez dos processos. O XP é o método ágil mais conhecido. O XP não é adequado para sistemas complexos de grande porte ou críticos, sendo mais indicado para o desenvolvimento de aplicações de pequeno porte, por exemplo, softwares para venda de produtos. Atualmente, as aplicações típicas são para ambientes Web.

Havia nas décadas de 1980 e início de 1990 uma visão geral de que a melhor maneira de se obter o software com a mais alta qualidade era em um contexto muito específico, formado por um cuidadoso planejamento de projeto, a garantia de qualidade formalizada, o uso de métodos de análise e de projeto apoiados por ferramentas CASE e controlados por um rigoroso processo de desenvolvimento de software.

Este cenário, na maioria das vezes, era aplicável a projetos de sistemas críticos que envolviam equipes grandes de empresas diferentes, geograficamente separadas. Quando envolviam projetos de sistemas de pequenas e médias empresas, a sobrecarga (*overhead*) envolvida na determinação do que deveria ser feito era maior do que o desenvolvimento propriamente dito. Da mesma forma, quando havia mudanças de requisitos, o retrabalho era necessário, mantendo-se a consistência com a especificação e o projeto do sistema.

A insatisfação com as abordagens ditas pesadas (ou tradicionais) levou na década de 1990 um número de desenvolvedores a propor novas abordagens, conhecidas como processos ágeis (do inglês *agile methodologies*), que resultaram no famoso Manifesto Ágil.<sup>11</sup> A proposta dos autores era desenvolver maneiras melhores de desenvolver software, que poderiam ser cumpridas desde que os indivíduos e a interação entre eles fossem mais importantes do que processos e ferramentas; o software em funcionamento mais do que a documentação abrangente, a colaboração com o cliente mais do que a negociação de contratos e responder a mudanças mais do que seguir um plano.

Entre os processos mais conhecidos, destacam-se o *Extreme Programming* (XP)<sup>12</sup> para desenvolvimento de software e o Scrum<sup>13</sup> para gerenciamento de projetos (veja mais detalhes na Seção 3.1).

O XP tem como base o desenvolvimento iterativo e o grande envolvimento do cliente. As suas principais características são:<sup>12</sup>

- a) Planejamento incremental.  
Planejamento inicial e refinamentos posteriores à medida que surjam novas informações.
- b) Pequenos entregáveis (*releases*).  
Desenvolvimento de pequenas versões do software para serem entregues com frequência aos clientes.
- c) Projeto simples.  
O projeto deve ser simples para possibilitar a entrega mais rápida.
- d) Desenvolvimento de testes antes do código (*test-first*).  
É mais fácil e rápido gerar o código se definir os seus testes antes.

- e) Refatoração (*refactoring*) frequente do código.  
A ideia é modificar sempre que possível a estrutura do código para facilitar a sua manutenção futura.
- f) Programação em pares.  
Dois programadores trabalhando juntos em um mesmo computador para melhorar a qualidade do código sem impactar os prazos de entrega.
- g) Propriedade coletiva do código.  
Isso permite que qualquer membro da equipe contribua com ideias para melhorar o código. Todos são responsáveis pelo código.
- h) Integração contínua do sistema.  
Todo código é integrado em um repositório continuamente. Isso melhora o reuso e compartilhamento de código e evita desperdício de esforços.
- i) Ritmo sustentável de trabalho.  
Para alcançar a qualidade necessária do código, os desenvolvedores devem estar motivados e completarem a iteração planejada.
- j) Cliente no local (*on-site*) em tempo integral.  
O cliente não só apoia os desenvolvedores, mas também faz parte da equipe. Isso melhora a comunicação da equipe e o entendimento do software.

A Figura 2.5 ilustra o processo de um entregável (*release*) ou uma pequena versão em XP.

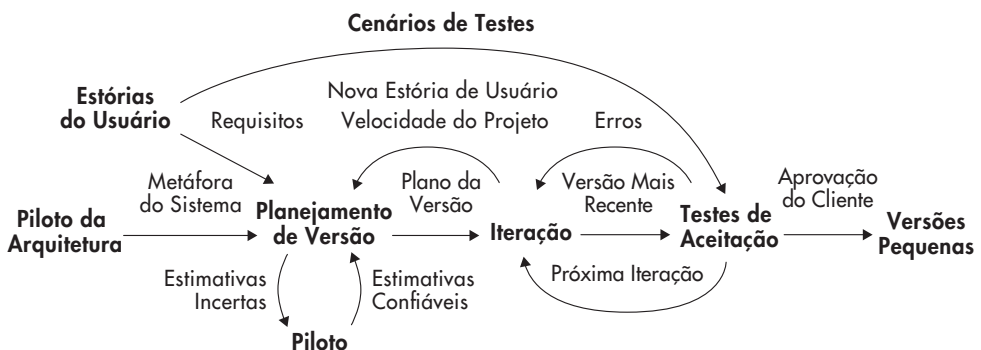


Figura 2.5 – Processo XP.<sup>14</sup>



Conforme indicado na Figura 2.5, o processo se inicia ao selecionar as histórias do usuário, que são os seus requisitos. Cada história descreve o que deve ser incluído no produto a ser desenvolvido. Em seguida, é realizado o Planejamento de Versão onde as histórias são priorizadas e divididas em tarefas. Para orientar o entendimento do funcionamento do sistema, um piloto de arquitetura (metáfora) do sistema é usado. Então, um plano da versão é definido, estabelecendo como será desenvolvida a versão do produto. Na iteração, o produto é desenvolvido e passa por Testes de Aceitação do cliente. Após a aprovação do cliente, o produto é liberado (Pequenas Versões). Após os Testes de Aceitação, uma nova versão pode ser implementada para novas histórias do usuário.

O XP não é adequado para sistemas complexos de grande porte ou críticos, sendo mais indicado para o desenvolvimento de aplicações de pequeno porte, como softwares para vendas de produtos. Atualmente, as aplicações típicas são para ambientes Web.

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

Uma questão recorrente é se os métodos ágeis tornam o processo de desenvolvimento realmente ágil. A agilidade não deve ser confundida com eliminação pura e simples de burocracia. Na falta de critérios ou de conhecimentos profundos da área desenvolvimento de software, acabam-se, muitas vezes, tomando decisões precipitadas que só serão percebidas após o software ser implantado e entrar na fase de manutenção.

Entretanto, a questão da agilidade não é resolvida apenas pela adoção das práticas do XP. A agilidade é alcançada a partir da manutenção dos valores e princípios dos métodos ágeis definidos no Manifesto Ágil.<sup>11</sup>

O XP e os métodos ágeis em geral têm sido erroneamente interpretados por não gerar documentação. Existe certamente uma documentação, porém somente a necessária. Escopos menores a cada versão do software podem ser entregues em prazos menores, porém a participação do cliente

no processo é fundamental para ter agilidade. O XP é ágil também, pois permite que mudanças do negócio sejam incorporadas mais rapidamente.

## EXERCÍCIOS

1. O que faz o XP ser ágil?
2. Uma das características importantes do XP é a intensa participação do cliente no desenvolvimento de software em tempo integral. Como isso poderia ser implementado na prática?
3. O XP depende de uma boa composição da equipe de projeto. Quais seriam os perfis dos seus membros para ter bom desempenho com o XP?
4. A interação entre os membros da equipe é muito importante no XP. Como se poderia incentivar essa interação?
5. É correto afirmar que o sucesso com XP depende de uma equipe pequena e coesa? Justifique.
6. Baseado na dinâmica do XP, como se poderia definir um projeto de software?
7. Qual seria um bom critério para definir as primeiras versões do software com XP?
8. Por que o XP não é adequado para o desenvolvimento de software de grande porte, por exemplo, um sistema de controle de metrô?
9. Por que o Manifesto Ágil representa uma quebra de paradigma em relação às abordagens tradicionais?
10. Comparando o processo Cascata com o do XP, em quais pontos são diferentes?

## SUGESTÕES DE LEITURA

BECK, Kent. Embracing change with extreme programming. *IEEE Computer*, vol. 32, issue 10, pp. 70-77, Oct., 1999.

BECK, Kent et al. *The agile manifesto*. 2001. Disponível em: <<http://agilealliance.org>>. Acesso em: 10 jun. 2011.

BECK, Kent; CYNTHIA, Andres. *Extreme programming explained: embrace change*. 2<sup>nd</sup> edition. USA: Addison-Wesley Professional, 2004.

BOEHM, Barry W. Get ready for agile methods, with care. *IEEE Computer*, vol. 35, issue 1, pp. 64-69, Jan., 2002.

COCKBURN, Alistair. Selecting a project's methodology. *IEEE Software*, vol. 17, issue 4, pp.64-71, Jul./Aug., 2000.

HIGHSMITH, Jim; COCKBURN, Alistair. Agile software development: the business of innovation. *IEEE Computer*, vol. 34, issue 9, 120-122, Sep., 2001.

REIFER, Donald J. How good are agile methods? *IEEE Computer* vol. 19, issue 4, pp. 16-18, Jul./Aug., 2002.

WELLS, James. D. *Extreme programming: a gentle introduction*. 1999. Disponível em: <<http://www.extremeprogramming.org>>. Acesso em: 15 jun. 2011.

WILLIAMS, Laurie; COCKBURN, Alistair. Agile software development: it's about feedback and change. *IEEE Software*, vol. 36, issue 6, pp. 39-43, Jun., 2003.

## Gerenciamento

---

As atividades de gerenciamento de projetos não fazem parte das atividades técnicas de desenvolvimento de software. Porém, ele afeta diretamente o sucesso de um projeto. Sucesso de projeto pode ser entendido como entrega nos prazos e nos custos estimados, uso adequado de recursos e grau de satisfação do cliente. Um projeto é definido como um esforço temporário conduzido para criar um único produto, serviço ou resultado.<sup>15</sup> Sendo o software o resultado de um esforço temporário de desenvolvimento, então essa iniciativa deve ser encarada como um projeto. Um projeto de software típico deve atender a muitos requisitos. É muito comum que, durante o andamento do projeto, os requisitos necessitem de mudanças. As atividades de gerenciamento de requisitos são fundamentais para o controle dessas mudanças.

---

### 3.1 GERENCIAMENTO DE PROJETOS

*A atividade de gerenciamento de projetos é fundamental para o sucesso de um desenvolvimento de software. O gerenciamento de projetos deve cuidar do dimensionamento do projeto, estabelecendo uma estratégia para atingir os resultados desejados dentro de restrições de custo e prazo. É uma atividade difícil, pois, muitas vezes, é baseada na experiência dos gerentes de projetos e em poucos dados de projetos anteriores. O resultado inicial*

*desta atividade é um Plano de Projeto, que deve ser usado para acompanhar e controlar o andamento do projeto. Conforme o projeto é realizado, replanejamentos podem ser necessários caso algum desvio significativo do projeto seja detectado. Uma boa referência para métodos e técnicas de gerenciamento de projetos é o PMBoK (Corpo de Conhecimentos em Gerenciamento de Projetos).<sup>15</sup>*

Esta seção não se refere a uma atividade típica de Engenharia de Software. Porém, é extremamente importante que o gerente de projetos saiba quais são as atividades que ele deve realizar para que o projeto atinja os seus objetivos. As atividades de gerenciamento caminham paralelas às atividades de desenvolvimento, existindo forte interação entre ambas.

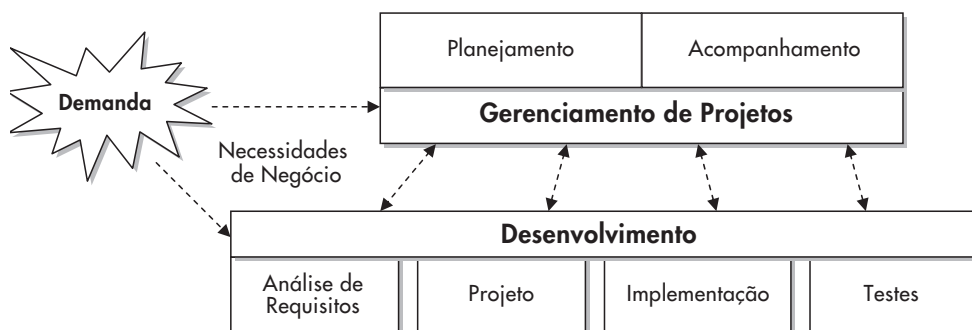
É um desafio muito grande para o gerente definir uma estratégia ou um plano de projeto para atender às necessidades do cliente. Uma necessidade do cliente pode chegar de várias formas. Uma das formas mais usuais é a área Comercial passar para a área de Desenvolvimento uma demanda já com algumas restrições de projeto predefinidos (prazo e custo), juntamente com a documentação técnica para apoiar o entendimento.

O gerente que elabora um plano de projeto deve levar em conta o escopo do projeto, os riscos, as restrições e premissas que nortearão a sua estratégia. O processo de desenvolvimento escolhido ajudará no dimensionamento das tarefas e dos recursos que serão aplicados. Uma atividade importante do planejamento de projeto é a definição de um cronograma. Outro item importante é o plano de comunicação do projeto, o qual permitirá controlar o fluxo de comunicação entre os interessados (também conhecidos como stakeholders) do projeto.

Uma vez que todos estes itens estejam definidos, o plano de projeto resultante deve ser aprovado por todos os interessados.

Ao iniciar o projeto, o plano é usado para acompanhar e controlar seu andamento e, em alguns casos, replanejamentos são necessários se desvios significativos forem detectados em relação ao planejado.

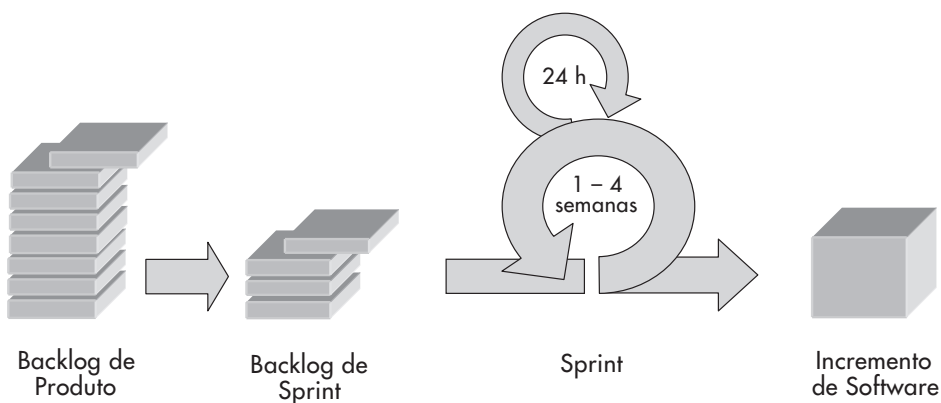
A Figura 3.1 representa a interação entre as atividades de gerenciamento de projetos tradicional e de desenvolvimento de software.



**Figura 3.1** – Interação entre gerenciamento de projetos tradicional e desenvolvimento de software.

As práticas de gerenciamento de projetos aplicáveis podem ser vistos no PMBoK (Corpo de Conhecimentos em Gerenciamento de Projetos), atualmente muito bem aceito na indústria de software.<sup>15</sup>

Acompanhando a evolução dos métodos ágeis discutidos na seção 2.5, com ênfase no processo XP, o processo para gerenciamento de projetos mais conhecido é o Scrum.<sup>13</sup> A Figura 3.2 ilustra o processo Scrum.



**Figura 3.2** – Processo Scrum.

O processo Scrum é um processo iterativo e incremental, dividido em dois ciclos principais. O primeiro ciclo chamado de Sprint, que tem uma duração entre 1 e 4 semanas, representa um período de uma iteração no qual se implementa uma funcionalidade a ser entregue aos *stakeholders*. Uma funcionalidade é definida dentro de um conjunto de características enviadas pelo cliente chamadas de *Backlog* de Produto. Estas são discutidas e priorizadas para constituírem o *Backlog* de Sprint que são as tarefas que devem ser realizadas. Nenhuma mudança é aceita durante um *Sprint*. Na data definida para o encerramento de um Sprint, uma reunião de revisão do produto obtido é realizada com os *stakeholders*. O ciclo se repete para o próximo *Sprint*.

O segundo ciclo é realizado dentro de um *Sprint*, onde uma reunião é realizada diariamente (24 horas) para medir o progresso do dia anterior, identificar eventuais problemas (impedimentos) e priorizar as tarefas do dia.

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

O gerenciamento é tão importante quanto o próprio desenvolvimento do software. Do mesmo modo, exige do profissional conhecimento e experiência nessa área. Há vários perfis e formações na indústria de software o que torna difícil ter um padrão quando se trata de boas práticas em gerenciamento. O PMBoK é um ótimo apoio para isso.

Muitas organizações implementaram escritórios de projetos (PMO – *Project Management Office*) para fazer gerenciamento de projetos de alto nível. A implementação de PMO pode melhorar a disciplina de gerenciamento de projetos.

As práticas do PMBoK podem ser aplicadas em diversos tipos e tamanhos de projetos de software.

Seguindo os métodos ágeis, o Scrum tem sido aplicado com sucesso em projetos de pequeno porte para equipes pequenas. O Scrum exige do

gerente de projeto uma disciplina muito rigorosa para conseguir cumprir um *Sprint* e entregar um produto finalizado.

## EXERCÍCIOS

1. Qual é a atividade de gerenciamento mais difícil para um gerente de projeto que acaba de receber um novo projeto? Justifique.
2. Como se poderia realizar um projeto se as demandas, tipicamente, chegam com prazos e custos predefinidos pela área Comercial?
3. O perfil de um gerente de projeto de software necessita ser técnico, por exemplo, um engenheiro? Justifique.
4. Alguns projetos de software fracassam pelo mau gerenciamento de projetos. Quais fatores contribuem para esse resultado?
5. Quais fatores são importantes para a montagem de uma equipe de projeto?
6. Em quais tipos de projeto o Scrum é mais aplicável? Justifique.
7. Como as mudanças solicitadas pelos *stakeholders* podem ser controladas durante a execução do Scrum?
8. Um bom planejamento deve identificar os riscos de um projeto. Quais riscos de um projeto com o Scrum poderiam ser considerados?
9. O que ocorrerá se ao fim de um *Sprint* o produto não for obtido?
10. Cite exemplos de aplicações de software que podem ser gerenciados com o Scrum.

## SUGESTÕES DE LEITURA

BERKUN, Scott. *A arte do gerenciamento de projetos*. Porto Alegre: Artmed, 2008.

PROJECT MANAGEMENT INSTITUTE. *Um guia do conjunto de conhecimentos em gerenciamento de projetos. PMBOK*. 4. ed. São Paulo: Project Management Institute, 2008.

ROYCE, Walker. *Software project management – a unified framework*. USA: Addison-Wesley Professional, 1998.

SCHWABER, Ken. *Agile Project Management with Scrum*. USA: Microsoft Press, 2004.



### 3.2. GERENCIAMENTO DE REQUISITOS

*Um projeto de desenvolvimento de software produz muitas informações. O esperado no resultado final é que o software atenda às necessidades do cliente dentro de restrições de custo e prazo. Assim é necessário que haja um gerenciamento destas informações desde a especificação de requisitos até a entrega do software ao cliente. O gerenciamento de requisitos trata não somente da entrega, mas também das mudanças que podem ocorrer nos requisitos. Ocorrem mudanças de requisitos ao longo do ciclo de desenvolvimento de software. Portanto, a atividade de gerenciamento de mudanças é fundamental para que os requisitos negociados e aprovados com cliente sejam controlados, atendendo-se às eventuais mudanças que venham a ocorrer. O gerenciamento de mudanças se dá com o rastreamento dos requisitos durante o desenvolvimento de software. O rastreamento permite analisar os impactos dessas mudanças ao processo de desenvolvimento.*

O gerenciamento de requisitos é essencialmente um processo para gerenciar as mudanças dos requisitos do sistema. As mudanças devem ser gerenciadas para assegurar que elas atendam a questões econômicas e contribuam para as necessidades de negócio da organização que adquire o sistema. A viabilidade das mudanças deve ser avaliada dentro do prazo e do orçamento do projeto.<sup>16</sup>

Segundo o MR-MPS,<sup>17</sup> o principal objetivo do gerenciamento de requisitos é controlar a evolução deles e tratar suas mudanças ao longo do desenvolvimento de software. Envolve também identificar os requisitos e os componentes do produto do projeto, bem como estabelecer e manter um acordo entre o cliente e a equipe de projeto sobre esses requisitos.

A parte central do processo de gerenciamento de requisitos é o gerenciamento de mudanças destes. Uma vez definidos os requisitos do software, eles são desenvolvidos ao longo do ciclo de desenvolvimento. Embora, estes requisitos tenham sido satisfatoriamente entendidos antes de iniciar a sua implementação, o fato é que eles irão mudar em algum momento do desenvolvimento. As causas dessas mudanças, invariavelmente são devidas

a erros cometidos durante o entendimento dos requisitos ou de maneira mais recorrente, a mudanças de ambiente, ou de negócio do cliente.<sup>7</sup>

Mais importante do que discutir as causas, é saber como minimizar os impactos de mudanças de requisitos no processo de desenvolvimento. As mudanças devem ser gerenciadas para que sejam aprovadas, revisadas e efetivamente implementadas, por meio de rastreamento de requisitos, análise de impactos e gerenciamento de configuração de software.<sup>7</sup> O gerenciamento de mudanças tem forte ligação com a configuração de software<sup>16</sup> (veja mais detalhes no Capítulo 7).

A análise de impactos de mudanças de requisitos tem por objetivo identificar de que forma uma mudança impacta nos planos do projeto que contêm as estimativas aprovadas de esforço e custo para os produtos de trabalho e tarefas, bem como os códigos de unidade ou módulos do software que necessitam ser modificados.<sup>17</sup>

Por essas análises, o responsável pelo gerenciamento do projeto é capaz de negociar com o cliente mudanças nos planos do projeto para atender às solicitações de mudanças de requisitos e, ao mesmo tempo, minimizar os riscos do projeto como desvios de cronograma e de custos.<sup>17</sup>

Para que o gerenciamento de mudanças de requisitos funcione, é fundamental entender o que é rastreamento de requisitos. Kotonya e Sommerville<sup>16</sup> dizem que rastreamento de requisitos está relacionado com a recuperação da fonte dos requisitos e o prognóstico dos efeitos dos requisitos. O rastreamento é fundamental para a análise de impactos quando os requisitos mudam. Assim, o gerenciamento de requisitos deve definir e manter sua rastreabilidade.

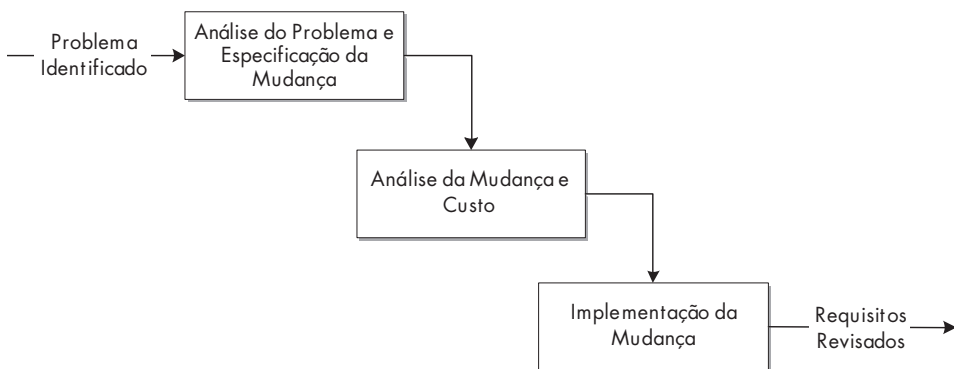
Segundo o IEEE,<sup>2</sup> a rastreabilidade é o grau em que o relacionamento pode ser estabelecido entre dois ou mais produtos do processo de desenvolvimento, especialmente produtos que tenham um relacionamento predecessor-sucessor ou mestre-subordinado com outro; por exemplo, o grau em que requisitos e projeto de um dado componente de software combinam.

Quando os requisitos são bem gerenciados, a rastreabilidade pode ser estabelecida, desde um requisito fonte, passando por todos os níveis de

decomposição do produto até seus requisitos de mais baixo nível e destes até o seu requisito fonte. Esse processo é conhecido como rastreabilidade bidirecional que auxilia a determinar se todos os requisitos fonte foram completamente tratados e se todos os requisitos de mais baixo nível podem ser rastreados para uma fonte válida.<sup>17</sup> Informação de rastreabilidade é usada para avaliar os impactos de uma solicitação de mudança. Ela contém informações sobre dependências, razões (por que foi especificado) e implementação de requisitos.<sup>16</sup>

O gerenciamento de mudanças refere-se aos procedimentos, processos e padrões que são usados para gerenciar mudanças nos requisitos do sistema. Assegura que informações similares sejam coletadas a cada proposta de mudança e que julgamentos completos são feitos sobre custos e benefícios das mudanças propostas.<sup>16</sup>

Um exemplo de gerenciamento de mudanças sugerido por Kotonya e Sommerville<sup>16</sup> pode ser visto na Figura 3.3. Inicialmente, um problema relacionado a alguma necessidade de mudança é identificado. O problema é analisado e uma especificação de mudança deve ser elaborada na forma de uma Solicitação de Mudanças. Em seguida, uma análise da solicitação é realizada com base nos impactos e nos custos envolvidos. Uma vez que o custo seja aceito pelo cliente, a mudança é implementada. Isso significa que os documentos de requisitos afetados são revisados e uma nova versão do documento é produzida.<sup>16</sup>



**Figura 3.3** – Gerenciamento de mudanças.<sup>16</sup>

Uma Solicitação de Mudanças pode ser rejeitada quando ela não é satisfatoriamente justificada, ou se os resultados das mudanças têm consequências inaceitáveis aos clientes, ou se os custos e o tempo requeridos são demasiadamente altos.<sup>16</sup>

Um importante mecanismo para realizar o gerenciamento de mudanças é uma matriz de rastreabilidade. Segundo o IEEE,<sup>2</sup> uma matriz de rastreabilidade permite registrar o relacionamento entre dois ou mais produtos de desenvolvimento. Alguns exemplos de matriz de rastreabilidade podem ser vistos nas Tabelas 3.1 e 3.2.

A Tabela 3.1 mostra um exemplo de rastreabilidade do tipo requisito-requisito onde os requisitos serão implementados em produtos de software. A matriz deve ser preenchida quando todos os requisitos forem negociados e acordados com o cliente no início do ciclo de desenvolvimento de software. Neste exemplo, o Req1 depende do Req2 e Req3, o Req2 depende do Req1 e Req3 e Req4 dependem dos Req1 e Req2. Se uma mudança no Req2 for solicitada, deve-se avaliar os impactos nos Req1, Req3 e Req4. Entretanto, se uma mudança no Req4 for proposta, não há impacto nos outros requisitos.

**Tabela 3.1** – Matriz de rastreabilidade do tipo Requisito-Requisito (adaptado)<sup>16</sup>.

Requisitos Reqs. Dependentes	Req1	Req2	Req3	Req4
Req1		X	X	
Req2	X		X	
Req3	X	X		
Req4	X	X		

A Tabela 3.2 apresenta outra matriz de rastreabilidade do tipo requisito-produto. Os produtos são desenvolvidos para implementar os requisitos em cada fase do ciclo de desenvolvimento de software. A Fase 1 gera Prod1 e Prod2, a Fase 2 gera Prod3 e Prod4 e, assim por diante. Cada produto

implementa um ou mais requisitos. A matriz deve ser preenchida à medida que os produtos são gerados. Por exemplo, o Prod1 implementa todos os requisitos, o Prod2 somente o Req2 e, assim por diante. Se uma mudança no Req2 for solicitada no início da Fase 3, deve-se analisar os impactos nos produtos Prod1, Prod2 e Prod4 que já existem. Os produtos Prod5 e Prod6 não existiriam neste momento. Se for uma mudança no Req3, somente o Prod1 será afetado e, portanto, deve-se analisar o impacto da mudança no Prod1.

**Tabela 3.2** – Matriz de rastreabilidade do tipo requisito-produto.

Produtos Requisitos	Fase 1		Fase 2		Fase 3	
	Prod1	Prod2	Prod3	Prod4	Prod5	Prod6
Req1	X		X			
Req2	X	X		X	X	
Req3	X					X
Req4	X		X			X

Os dois tipos de matriz de rastreabilidade podem ser combinados para complementar as análises de impactos das mudanças de requisitos solicitadas.

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

Um desenvolvimento de software gera e deve controlar uma grande quantidade de informações. O gerenciamento de requisitos é essencialmente um processo com este fim e permite que se garanta que estas informações sejam entregues às pessoas certas e no tempo certo.<sup>16</sup> Pode-se dizer, também, que se um cliente não está satisfeito com a qualidade das entregas significa que houve problemas com o gerenciamento de requisitos.

Um projeto de desenvolvimento de software não é isento de mudanças. Toda mudança é particularmente danosa para os objetivos de um projeto. É necessário, então, ter uma estratégia diferente do gerenciamento de projetos que possa cuidar exclusivamente da evolução dos requisitos.

Nem sempre uma mudança é viável, mas a análise de impactos baseada em registros de uma matriz de rastreabilidade, permite argumentar com os clientes sobre as consequências no processo de desenvolvimento de software e nos custos do projeto.

## EXERCÍCIOS

1. Quais são as informações de entrada para o gerenciamento de requisitos?
2. Quais são as consequências da falta de um processo de gerenciamento de requisitos?
3. Por que o gerenciamento de mudanças é a atividade central do processo de gerenciamento de requisitos?
4. Em sua opinião, por que os requisitos necessitam ser mudados durante o ciclo de desenvolvimento de software?
5. Considere que uma Solicitação de Mudança deve ser atendida. Sugira formas de avaliar se a mudança foi implementada tal qual foi analisada e aprovada.
6. Em que situações uma Solicitação de Mudanças deveria ser rejeitada?
7. Preencha a matriz de rastreabilidade do tipo requisito-requisito de um projeto que você esteja participando.
8. Como se deve preencher a matriz de rastreabilidade do tipo requisito-produto? Exemplifique.
9. Por que as duas matrizes de rastreabilidade apresentadas podem dar informações complementares para a análise de impacto de mudanças?
10. Faça um levantamento de projetos que você tenha participado e que não obtiveram sucesso (veja definição de sucesso de projeto na seção 1.2). Em seguida, elabore uma tabela destacando as causas deste insucesso entre a falta de gerenciamento de projetos ou a falta de gerenciamento de requisitos ou ambos.

## SUGESTÕES DE LEITURA

CHRISSIS, Mary B.; KONRAD, Mike; SHRUM, Sandra. *CMMI for development – guidelines for process integration and product improvement*. 3<sup>rd</sup> edition. USA: Addison-Wesley Professional, 2011.

KOTONYA, Gerald; SOMMERVILLE, Ian. *Requirements engineering: process and techniques*. New York: Wiley, 1998.


SOFTEX. *Guia de implementação – parte 1: fundamentação para implementação do nível G do MR-MPS*. Softex, Julho, 2011.

SOFTEX. *MPS.BR – guia geral*. Versão 2011. Softex, Junho, 2011.

SOMMERVILLE, Ian. *Engenharia de Software*. Tradução: Kalinka Oliveira e Ivan Bosnic. Revisão Técnica: Kechi Hirma. 9. ed. São Paulo: Prentice Hall, 2011.

# Desenvolvimento de software

---

 software é um produto complexo. Sendo complexo, as atividades de desenvolvimento devem ser bem definidas para a equipe de desenvolvedores. O desenvolvimento de software pode ser comparado à construção de uma casa. Necessitam-se um arquiteto, empreiteiro, pedreiros, eletricitas, encanadores etc. Não se imagina, por exemplo, que um pedreiro vá assentando os tijolos sem ter uma ideia da casa concebida por um engenheiro civil. Assim, existem atividades que devem ser realizadas, sem as quais haverá desperdício de material e tempo. Da mesma forma, existem atividades de desenvolvimento de software.

---

## 4.1. ENGENHARIA DE SISTEMAS

*A atividade de Engenharia de Sistemas tem por objetivo entender as necessidades de negócio do cliente e especificar os requisitos do sistema computacional, incluindo-se os requisitos do software em alto nível. Uma das tarefas realizadas é a Prova de Conceito, cuja finalidade é verificar se o sistema é exequível, levando-se em conta os riscos, recursos e tecnologias aplicáveis. A outra tarefa é identificar as funções, restrições e o desempenho do sistema. Uma forma de visualizar as funções do sistema computacional é identificar os seus elementos constituintes por meio de um modelo hierárquico. Cada elemento pode ser, em si, um sistema computacional completo com hardware, software, banco de dados e pessoas.*



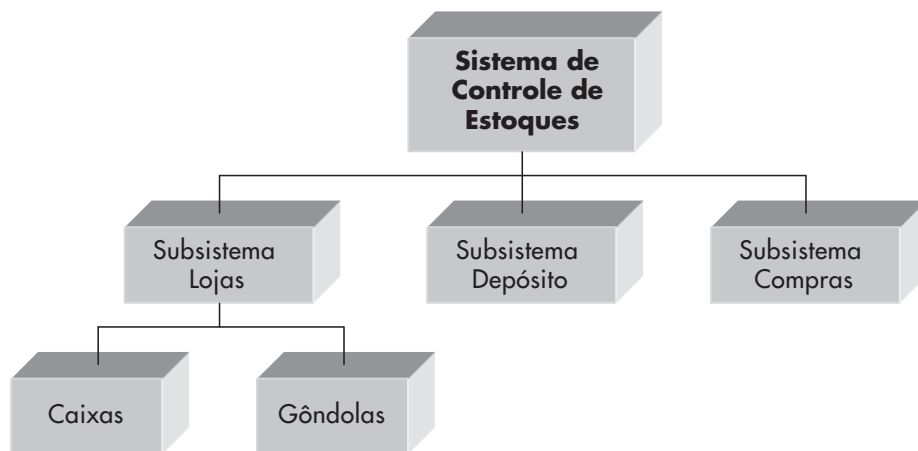
A atividade de Engenharia de Sistemas tem por objetivo entender as necessidades de negócio do cliente e estabelecer requisitos do ponto de vista sistêmico ou de um sistema computacional (hardware, software, banco de dados e de pessoas) até definir em alto nível o que será desenvolvido em software. O elemento pessoas representa um aspecto muito importante dos sistemas atuais onde uma pessoa pode ser responsável por uma parte da funcionalidade do sistema.

Uma tarefa inicial dessa atividade é realizar uma Prova de Conceito (PoC – Proof of Concept) para verificar se o sistema é exequível, levando-se em conta os riscos, os recursos e as tecnologias aplicáveis. Para atender aos objetivos de uma PoC, um protótipo pode ser desenvolvido.

Outra tarefa é identificar as funções, as restrições e o desempenho do sistema. Um sistema é constituído de elementos que formam um conjunto coeso cujo objetivo é executar um método, procedimento ou controle ao processar dados. Para sistemas complexos, um elemento por si pode ser considerado um sistema completo. Então, uma das formas mais usadas para representar um sistema é um diagrama hierárquico no qual o nível superior é representado pelo próprio sistema e os níveis inferiores pelos subsistemas ou elementos constituintes mais simples. A Figura 4.1 ilustra essa visão de sistema.

O Sistema de Controle de Estoques é constituído dos subsistemas Lojas, Depósito e Compras. O subsistema Lojas, por sua vez, é constituído de Caixas e Gôndolas.

Para cada um desses subsistemas deve-se alocar um hardware, software, banco de dados ou de pessoas, respeitando-se as funções, o desempenho e as restrições do sistema. Assim, pode-se imaginar que algumas disciplinas poderiam ser necessárias para o desenvolvimento do sistema: engenharia de hardware, engenharia de software, engenharia de banco de dados e engenharia de usabilidade, respectivamente.



**Figura 4.1** – Hierarquia de sistema.

O documento típico dessa atividade é a Especificação de Sistema, que é constituída de uma visão geral do sistema, sua descrição funcional, a descrição dos subsistemas (com identificação de hardware, software, banco de dados e pessoas) e modelos do sistema.

Outro documento bem conhecido é o Documento de Visão do RUP que descreve o produto do ponto de vista dos *stakeholders*. Definido originalmente para processo de software, ele pode ser usado para especificação do sistema onde se destacam as características do produto. O Documento de Visão é constituído de declaração do problema e da solução, descrição dos *stakeholders*, ambiente do usuário, visão geral e características do sistema.

O foco deste livro está relacionado com o desenvolvimento de software (Engenharia de Software). Assim, embora importantes, as demais disciplinas ligadas ao hardware, banco de dados e pessoas não são tratadas aqui.

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

A visão integrada das várias partes que constituem um sistema é difícil no início, porém é fundamental para a etapa de integração do sistema quando as partes forem desenvolvidas. Vários sistemas apresentam

problemas de interface que atrasam as entregas e comprometem os custos do projeto, devido à falta desse tratamento sistêmico.

Este conceito é necessário em vários sistemas atuais. Por exemplo, sistemas críticos (industriais, comerciais, financeiros, de transporte, aeronáuticos, espaciais), sistemas de tempo real, sistemas de software intensivos etc. são candidatos naturais. Devido à complexidade desses sistemas, um de seus elementos pode ser considerado um sistema computacional completo.

## EXERCÍCIOS

1. O que é um sistema?
2. Um aspecto importante na definição de um sistema computacional são as suas restrições. O que você entende por restrição de um sistema?
3. Dê exemplos de sistemas onde o elemento pessoas é responsável por uma parte das funções de um sistema computacional.
4. A PoC pode ser realizada para verificar os custos e prazos de um sistema? Justifique.
5. Quando um protótipo é aplicável na atividade engenharia de sistemas?
6. Um diagrama hierárquico pode ser usado para representar o sistema e seus elementos constituintes. Elabore um diagrama hierárquico de um sistema conhecido.
7. Se for representar o mesmo sistema do exercício anterior por meio de uma descrição narrativa, quais são as vantagens e as desvantagens em relação a um diagrama hierárquico?
8. A Engenharia de Sistemas é a primeira atividade técnica do desenvolvimento de sistemas. Porém, necessita-se ter um bom conhecimento do domínio do problema (ou do negócio) para essa atividade. Qual seria o perfil adequado da equipe para esta atividade?
9. Quais são as funções básicas de um sistema de comércio eletrônico, por exemplo, de livros? Faça uma lista destas funções.
10. Aloque as funções identificadas no exercício anterior em hardware, software, banco de dados e pessoas.

## SUGESTÕES DE LEITURA

PFLEEGER, Shari L. *Software engineering: theory and practice*. 2<sup>nd</sup> edition. New Jersey: Prentice Hall, 2001.

PRESSMAN, Roger S. *Software engineering – a practitioner's approach*. 6<sup>th</sup> edition. New York: McGraw Hill, 2007.

SOMMERVILLE, Ian. *Engenharia de Software*. Tradução: Kalinka Oliveira e Ivan Bosnic. Revisão Técnica: Kechi Hirama. 9. ed. São Paulo: Prentice Hall, 2011.

## 4.2. ANÁLISE

*A atividade de análise tem por objetivo entender os requisitos do sistema e detalhar os requisitos do software. Portanto, o foco é o refinamento dos requisitos de sistema em requisitos de software. Essencialmente, a atividade de análise deve responder à seguinte questão: "O que é o software?". Para isso, o analista de sistema pode usar algumas representações para comunicar o seu entendimento. Existem basicamente duas abordagens para isso: Estruturada e Orientação a Objetos. Cada abordagem é definida por métodos tais como DeMarco,<sup>18</sup> Gane & Sarson,<sup>19</sup> e Yourdon<sup>20</sup> na Estruturada, e OMT de Rumbaugh et al.,<sup>21</sup> OOSE de Jacobson et al.<sup>22</sup> e Booch<sup>23</sup> na Orientação a Objetos. Atualmente, a UML (Unified Modeling Language)<sup>24, 25</sup>, que na realidade é uma linguagem de modelagem e não um método em si, acabou se tornando uma referência na indústria de software para representação deste.*

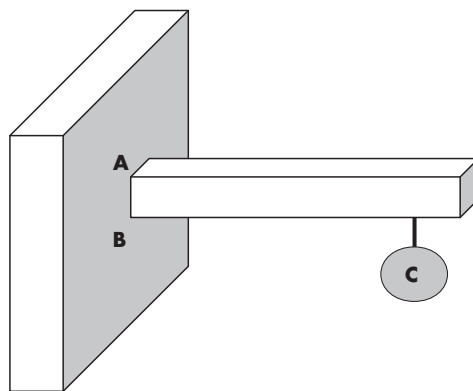
A atividade de Análise tem por objetivo entender os requisitos do sistema e detalhar os requisitos do software. Com base nas definições da atividade de Engenharia de Sistemas, nesta atividade, deve-se compreender o domínio da informação, a funcionalidade requerida, o desempenho e as interfaces exigidas.

O foco nesta atividade é o refinamento dos requisitos de sistema em requisitos de software. Para isso, deve-se reconhecer os elementos básicos do problema, avaliar a solução considerando os eventos, as interfaces, as funções e as restrições de projeto, sintetizar algumas soluções, criar modelos para melhorar a compreensão do fluxo de informações e o comportamento do software e, quando aplicável, implementar protótipos funcionais.

Na atividade de Análise deve-se responder à seguinte questão: “O que é o software?”; e não, “Como o software será desenvolvido?”. Esta diferenciação é importante, pois a tendência dos desenvolvedores (analistas, arquitetos e programadores) é dar soluções para o problema. Como se poderiam dar soluções se o problema ainda não foi compreendido? Por exemplo, uma função do software é verificar os dados do cliente e não buscar dados do cliente, comparar os dados com os dados fornecidos e exibir o status (existe ou não dados) do cliente.

Nessa atividade, o analista de sistemas deve ter uma comunicação intensa com o cliente para capturar os requisitos do software. Esta comunicação pode ser feita por meio de reuniões (JAD – *Joint Application Development*), fazendo uso ou não de representações (diagramas), de protótipos etc. O objetivo é obter o melhor entendimento possível do problema.

Nas representações de software, é importante que se utilizem elementos mais próximos do domínio do problema (industrial, financeiro, comercial etc.) representado, que não necessitem de textos descritivos para seu correto entendimento. Por exemplo, a Figura 4.2 adaptada da obra de Galileu de 1638, tinha um intuito bem definido: o de representar um método de análise de força de uma viga que havia sido proposto pelo cientista.<sup>26</sup> Ou seja, a partir de um objeto de uma dada massa no ponto C, quais são as forças agentes nos pontos A e B, levando-se em conta o comprimento da viga?



**Figura 4.2** – Representação adaptada de Galileu do método de análise proposto (adaptada).<sup>26</sup>

Para a representação do software pode-se usar diversos métodos disponíveis na literatura. Existem duas abordagens mais conhecidas: Estruturada e Orientada a Objetos (veja mais detalhes na Seção 1.4 e a seguir).

Uma vez obtidas as funções do software, pode-se desenvolver uma Especificação de Requisitos de Software. Essa especificação pode ser elaborada usando-se a norma IEEE 830-1984<sup>27</sup> que apresenta basicamente uma visão geral do sistema, requisitos específicos e informações complementares. A especificação pode ser acompanhada de representações do software e um protótipo funcional executável que serve para a validação dos requisitos exigidos pelo usuário ou um Manual do Usuário (preliminar) que serve para verificar o software do ponto de vista de usabilidade.

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

Talvez seja a Análise a atividade mais difícil de ser realizada. Não pelas técnicas existentes, que são relativamente simples, mas pela compreensão do que realmente se espera obter com essa atividade. Alguns analistas acabam curto-circuitando as atividades com as de Projeto e de Implementação, por falta dessa compreensão.

Também não se trata apenas de criar representações ou modelos usando abordagens conhecidas. As técnicas são meios pelos quais se consegue ter a melhor compreensão dos requisitos funcionais e não funcionais do software. Portanto, não se pode confundir o resultado físico com os objetivos da Análise.

Naturalmente, como em qualquer atividade de engenharia, supõe-se que os responsáveis dominem plenamente os métodos e as técnicas para obter resultados com qualidade.

Deve-se despende um tempo adequado para a atividade Análise. Caso isso não seja observado, as consequências típicas são retrabalhos em artefatos já desenvolvidos e aumentos não previstos de prazos e custos.

## EXERCÍCIOS

1. Muitos desenvolvimentos de software partem de uma Análise breve. Quais são as consequências desta brevidade?
2. É possível não realizar a atividade de Análise em um desenvolvimento de software? Justifique.
3. Qual é a importância da especificação baseada em um padrão (por exemplo, IEEE 830)<sup>27</sup> para o projeto de software?
4. Quais são as dificuldades de realizar a atividade de Análise no início do desenvolvimento de software?
5. O entendimento do domínio do problema exige conhecimento do negócio do cliente. O que é necessário para facilitar esse entendimento?
6. Qual é a vantagem de usar uma representação gráfica de um software na atividade de Análise?
7. Por que é difícil representar o software, se comparado ao hardware?
8. Por que, muitas vezes, um programador não é adequado para realizar as atividades de Análise?
9. A atividade de Análise exige um esforço de abstração para definir o problema ("O que é o software?"). Porém, acaba-se definindo a solução ("Como será desenvolvido?"). Por que isso ocorre?
10. Por que desenvolver somente os modelos e as representações de software exigidos não atinge os objetivos da Análise?

## SUGESTÕES DE LEITURA

PFLEEGER, Shari L. *Software engineering: theory and practice*. 2<sup>nd</sup> edition. New Jersey: Prentice Hall, 2001.

PRESSMAN, Roger S. *Software engineering – a practitioner's approach*. 6<sup>th</sup> edition. New York: McGraw Hill, 2007.

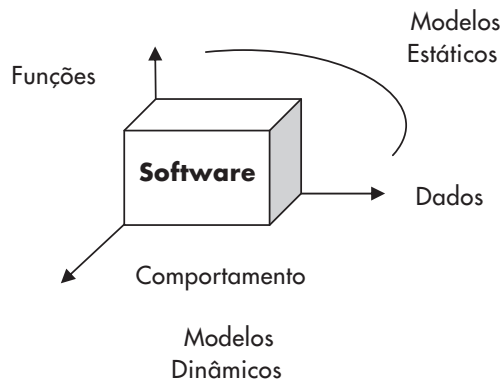
SOMMERVILLE, Ian. *Engenharia de Software*. Tradução: Kalinka Oliveira e Ivan Bosnic. Revisão Técnica: Kechi Hiramã. 9. ed. São Paulo: Prentice Hall, 2011.

### 4.2.1. Análise estruturada

*A análise estruturada é a abordagem mais antiga e mais conhecida de representação de software. Um dos fatores para o seu sucesso é a maneira organizada com que trata a complexidade de um software. Inicia com uma visão geral do sistema de software que interage com o seu ambiente e, em seguida, detalha as funções do software até o nível de funções elementares. Um aspecto importante desta organização é o controle do escopo do software à medida que ele vai sendo detalhado. Assim, softwares de qualquer complexidade funcional podem ser modelados por análise estruturada. Uma das ferramentas mais usadas para a modelagem de sistemas é o Diagrama de Fluxo de Dados (DFD).*

Na abordagem Estruturada, o foco principal é a representação das funções do software. Podem-se dividir as representações em modelos estáticos (apresentam a estrutura das funções e dados do software) e modelos dinâmicos (apresentam o comportamento do software), conforme a Figura 4.3.

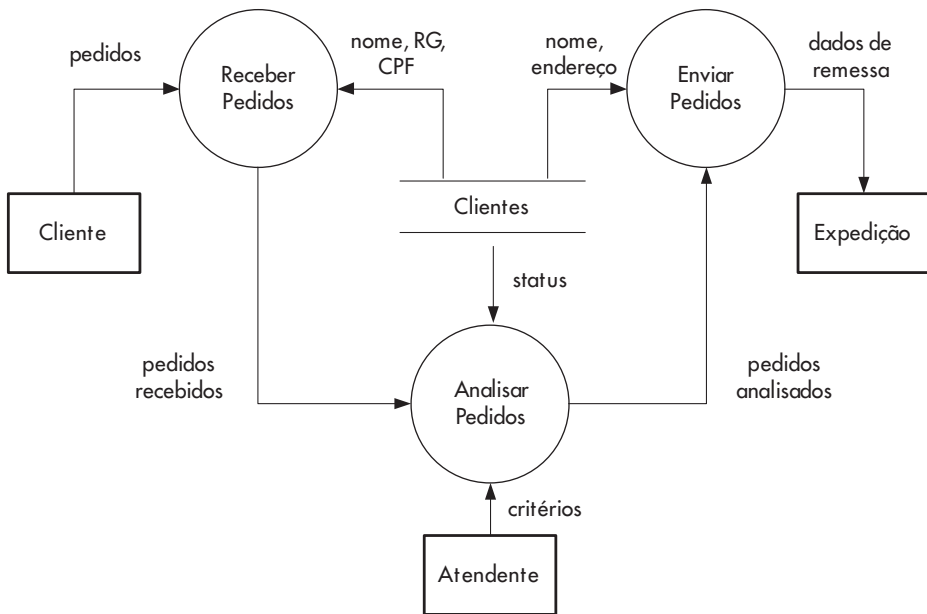
Um desses modelos pode ser visto na Figura 4.4, que é um Diagrama de Fluxo de Dados, conhecido como DFD, que é um dos modelos estáticos da abordagem Estruturada. O DFD é uma das ferramentas mais usadas para modelagem de sistemas, principalmente para sistemas em que as funções são de fundamental importância e mais complexas do que os dados manipulados pelo sistema.



**Figura 4.3** – Modelos estáticos e dinâmicos do software.



A Figura 4.4 representa um Sistema de Venda de Produtos genérico. Para a modelagem do sistema, inicia-se com uma visão geral do sistema de software interagindo como o seu ambiente e, em seguida, detalha as funções do software até o nível de funções elementares. Um aspecto importante desta organização é o controle do escopo do software à medida que ele vai sendo detalhado.



**Figura 4.4** – DFD típico de um Sistema de Venda de Produtos.

Na Figura 4.4 estão representados todos os componentes de um DFD:

- Funções do software: Receber Pedidos, Analisar Pedidos e Enviar Pedidos.
- Repositórios de dados: Clientes.
- Dados de entrada e saída das funções: pedidos; nome, RG, CPF; nome, endereço; dados de remessa; pedidos recebidos; status; pedidos analisados, critérios.
- Entidades externas que interagem com o sistema de software: Cliente, Atendente e Expedição.

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

A análise estruturada tem como objetivo definir funções, dados e comportamento do software, com foco preferencial nas funções. Os modelos estáticos e dinâmicos correspondentes devem ser desenvolvidos, mas não há necessariamente uma ordem predefinida para criar estes modelos. A ordem vai depender do tipo de aplicação que indica a direção a ser tomada.

Caso o software se trate fortemente de uma aplicação de banco de dados, a modelagem deverá ser iniciada pela dimensão dados. Caso se trate fortemente de uma aplicação de tempo real, a modelagem do comportamento deverá ser priorizada. Em casos gerais, a modelagem das funções em alto nível deverá ser sempre realizada. Em seguida, o software deverá ser modelado nas três dimensões em detalhes.

Uma característica importante da modelagem com DFD é que o modelo obtido não representa uma sequência de ativações de funções do software. As funções não estão ligadas por um aspecto temporal. Isoladamente, cada função representada define o seu objetivo, as entradas consumidas e as saídas produzidas.

## EXERCÍCIOS

1. Por que o DFD é definido como um modelo estático?
2. De que maneira o DFD trata a complexidade de um software?
3. Por que o DFD, em geral, pode ser o modelo inicial para todos os desenvolvimentos de software?
4. Por que uma das dimensões do software deve ser priorizada, dependendo do tipo de aplicação a ser desenvolvida?
5. Por que o DFD é diferente de um fluxograma?
6. Por que o DFD não pode ser confundido como um programa em alto nível?
7. O DFD é detalhado a partir do seu nível mais alto de abstração, conhecido como nível de Contexto, que representa o software e as entidades externas do ambiente com as quais interage. Qual é a importância do nível de Contexto?

8. Por que se diz que o escopo do software pode ser mantido com o DFD?
9. Por que em aplicações de tempo real, a representação do seu comportamento deve ser priorizada em relação às demais dimensões?
10. Cite um exemplo de aplicação, onde a modelagem de dados deve ser priorizada.

### SUGESTÕES DE LEITURA

DeMARCO, Tom. *Análise estruturada e especificação de sistemas*. Rio de Janeiro: Campus, 1989.

GANE, Chris; SARSON, Trish. *Análise estruturada de sistemas*. Rio de Janeiro: LTC, 1983.

McMENAMIM, Stephen M.; PALMER, John F. *Análise essencial de sistemas*. São Paulo: McGraw-Hill, 1991.

YOURDON, Edward. *Análise estruturada moderna*. Rio de Janeiro: Campus, 1990.

#### 4.2.2. Análise Orientada a Objetos

A análise Orientada a Objetos usa o conceito de classe e objeto para a modelagem do software. Foram publicados diversos métodos que apóiam a Orientação a Objetos sendo os mais conhecidos OMT,<sup>22</sup> OOSE<sup>21</sup> e Booch<sup>23</sup>. Estes métodos propõem diversos modelos de representação do software. Com a padronização da UML (Unified Modeling Language)<sup>24, 25</sup> pela OMG (Object Management Group) todos os modelos propostos por estes métodos foram unificados em um único conjunto de representações. Atualmente a UML é uma referência na indústria de software.

Na abordagem Orientação a Objetos, os elementos centrais de modelagem são a classe e o objeto. Do mesmo modo que a abordagem Estruturada, diversos métodos foram desenvolvidos. Os mais conhecidos são:

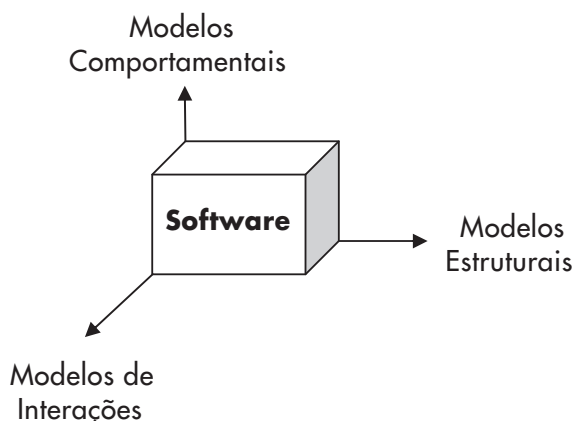
- a) O OMT (do inglês *Object-Modeling Technique* ou técnica de modelagem de objetos), que se baseia em três tipos de modelos: (1) modelo de objetos, que descreve as informações estáticas; (2) modelo dinâmico, que representa as

transições e (3) modelo funcional, cujo foco são os processos e o fluxo de informação do software.<sup>22</sup>

- b) O OOSE (do inglês *Object-Oriented Software Engineering* ou Engenharia de Software Orientada a Objetos), no qual o conceito de Caso de Uso foi desenvolvido para dar uma visão funcional de alto nível ao software.<sup>21</sup>
- c) O método proposto por Booch define a integração de um microprocesso para identificar e especificar as classes e os objetos e um macroprocesso que segue um processo tradicional, como o Cascata, para desenvolver e manter o software.<sup>23</sup>

Esses métodos foram predecessores da UML, que acabou se tornando uma referência na indústria de software para a representação de software.<sup>24, 25</sup> Atualmente, a UML está na sua versão 2.4 e sob a responsabilidade da OMG.

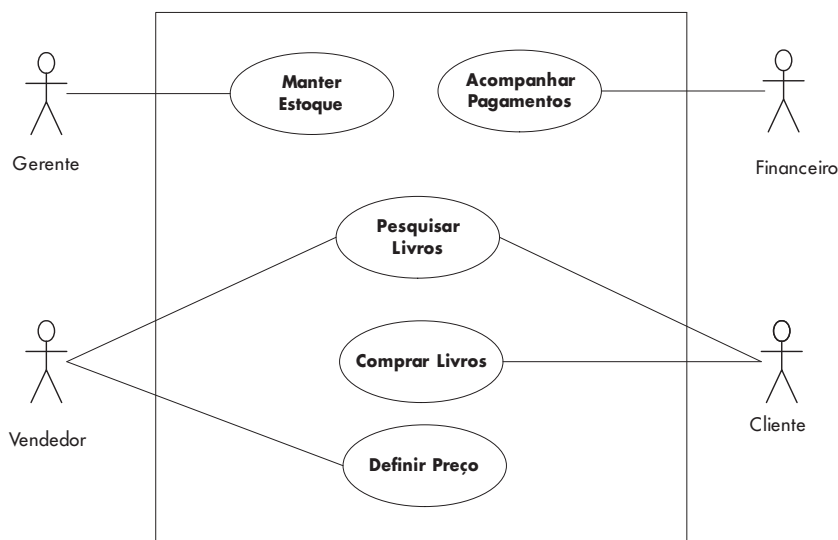
A versão 2.4 da UML define grupos de modelos chamados comportamentais, estruturais e de interações, conforme apresentado na Figura 4.5. Os modelos comportamentais (por exemplo, diagrama de casos de uso) representam construções funcionais; os estruturais (por exemplo, diagrama de classes) representam construções estáticas e os de interações (por exemplo, diagrama de sequência) definem construções dinâmicas do sistema.



**Figura 4.5** – Modelos comportamentais, estruturais e de interações do software da UML.

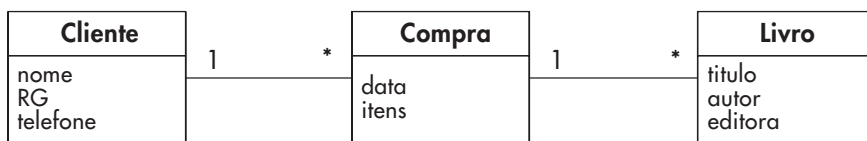
As Figuras 4.6, 4.7 e 4.8 apresentam um exemplo de cada um dos modelos aplicando a UML.

A Figura 4.6 apresenta um Diagrama de Casos de Uso típico de um sistema de venda de livros. Estão representados os atores que interagem com o sistema e os casos de uso representados pelas elipses, que são as funções do sistema. O retângulo representa a fronteira entre o ambiente externo e o sistema.



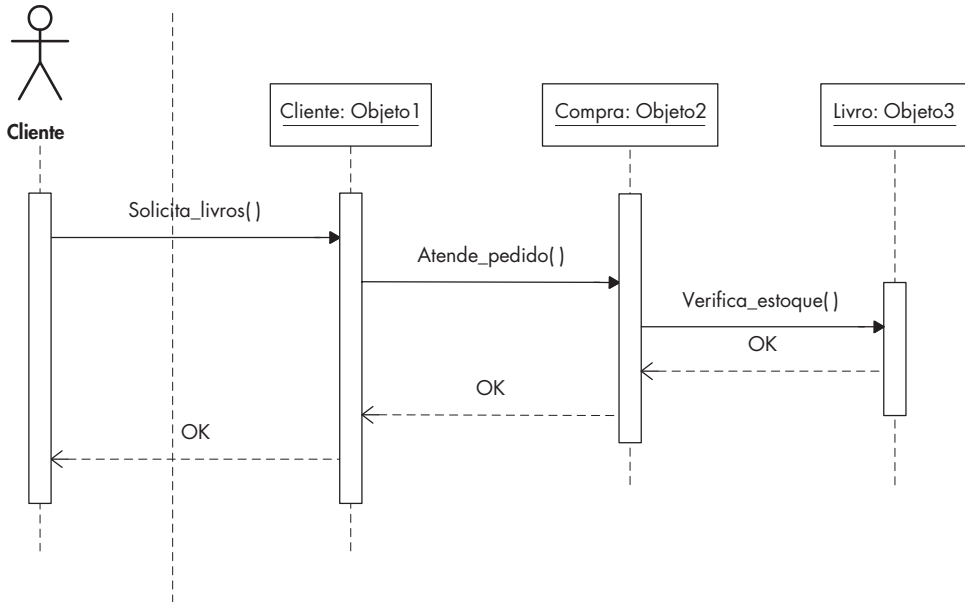
**Figura 4.6** – Diagrama de Casos de Uso de um sistema de venda de livros.

A Figura 4.7 apresenta um Diagrama de Classes simplificado para o caso de uso *Comprar Livros*. As classes presentes nesta figura, *Cliente*, *Compra* e *Livro*, estão relacionadas de maneira que o cliente possa realizar uma compra de livro no exemplo um (1) cliente faz muitas (\*) compras e uma (1) compra tem muitos (\*) livros.



**Figura 4.7** – Diagrama de Classes para o caso de uso: *Comprar Livros*.

A Figura 4.8 representa um Diagrama de Sequência para o caso de uso *Comprar Livros* usando os objetos das classes apresentadas na Figura 4.7. A sequência de mensagens entre os objetos permite que uma transação de compra seja realizada.



**Figura 4.8** – Diagrama de Sequência para o caso de uso: *Comprar Livros*.

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

Diferentemente da análise estruturada, a análise Orientada a Objetos, com UML, inicia-se com alguns modelos comportamentais, sendo o Diagrama de Caso de Uso o artefato a ser desenvolvido. Em seguida, alguns modelos estruturais (por exemplo, Diagrama de Classes) e finalmente, modelos de interação (por exemplo, Diagrama de Sequência) devem ser desenvolvidos. A ideia presente nesta ordem é que estabelecer, inicialmente, a funcionalidade e o escopo do software é mais adequada do que iniciar pela sua estrutura ou interação entre seus elementos.

Durante muitos anos, anteriores ao surgimento da UML, os desenvolvimentos de software orientados a objetos estavam centrados em programação, por exemplo, com linguagem C++. Logo se percebeu que havia dificuldades em gerenciar projetos pela falta de visão do escopo do software. Em outras palavras, o desenvolvimento baseado em classes e objetos exigia um esforço de abstração razoável para estabelecer as fronteiras do software.

O trabalho de Jacobson et al.<sup>22</sup> foi decisivo para quebrar esta dificuldade, que contribuiu decisivamente para o desenvolvimento da UML. A sua contribuição foi o conceito de Caso de Uso que propiciou a introdução de uma visão funcional de alto nível para o desenvolvimento de softwares orientados a objetos. Em essência, os diagramas de Caso de Uso são muito parecidos com os diagramas DFD de alto nível.

A UML tem sido aplicada com sucesso em processos evolutivos, iterativos e incrementais.

## EXERCÍCIOS

1. Por que é difícil definir o escopo do software a partir de classes e objetos do sistema?
2. Por que é mais fácil trabalhar com classes e objetos a partir de Casos de Uso?
3. O que representa o Diagrama de Sequência?
4. Suponha que um cliente encomendou uma aplicação e você terá de iniciar o entendimento de suas necessidades para desenvolver o software. Como os casos de usos podem ser identificados?
5. Elabore um diagrama de classes a partir do caso de uso *Pesquisar Livros* contido nesta seção.
6. Elabore um Diagrama de Sequência para este exemplo.
7. O conceito de "Caso de Uso" pode ser considerado como pertencente à abordagem Orientada a Objetos? Justifique.
8. O objetivo da atividade de Análise é definir "O que é o software?". Por que os modelos discutidos nesta seção atendem a este objetivo?

9. Um ator no Diagrama de Caso de Uso representa uma entidade externa com a qual o sistema interage. Dê um exemplo de ator que não seja uma pessoa ou uma área organizacional.
10. Quais são as semelhanças e diferenças entre o Diagrama de Caso de Uso e o diagrama DFD discutido anteriormente?

### SUGESTÕES DE LEITURA

BOOCH, Grady. *Object-oriented analysis and design with applications*. 2<sup>nd</sup> edition. USA: Addison Wesley Professional, 1994.

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. *The unified modeling language – user guide*. 2<sup>nd</sup> edition. USA: Addison-Wesley Professional, 2005.

JACOBSON, Ivar; CHRISTERSON, Magnus; JONSSON, Patrik; ÖVERGAARD, Gunnar. *Object-oriented software engineering: a use case driven approach*. USA: Addison-Wesley Professional, 1992.

RUMBAUGH, James; BLAHA, Michael; PREMERLANI, William; EDDY, Frederick; LORENSEN, William. *Object-oriented modeling and design*. Englewood Cliffs: Prentice-Hall, 1991.

## 4.3. PROJETO

A atividade de Projeto tem por objetivo estabelecer uma arquitetura do software que seja realizável. Nesta atividade, algumas decisões de projeto são tomadas para atender aos requisitos não funcionais tais como desempenho, confiabilidade e manutenibilidade. Diferentemente da Análise, no Projeto a questão a ser respondida é: “Como o software será desenvolvido?”. A resposta é a arquitetura do software representada por um modelo de arquitetura, que podem ser do tipo modelo baseado em repositório, modelo cliente-servidor, modelo em camadas ou suas combinações. Uma vez definido um modelo de arquitetura do sistema, a próxima atividade é a decomposição do sistema em módulos. Um módulo é um componente de sistema que fornece serviços para outros componentes, mas não é normalmente considerado como um sistema separado, como é o caso de um subsistema.



Após a Especificação de Requisitos de Software, complementada por diagramas e protótipos da atividade de Análise, a atividade de Projeto representa a ligação entre a codificação do software e os seus requisitos. A atividade de Projeto visa estabelecer uma arquitetura do software que seja realizável, apoiada em definições de módulos funcionais, detalhamento dos procedimentos ou algoritmos e interfaces entre os módulos.

A arquitetura de software é a estrutura ou as estruturas do sistema que abrangem os elementos de software, as suas propriedades (atributos) visíveis externamente e seus relacionamentos. A arquitetura é resultado de um conjunto de decisões técnicas e de negócio. Essas decisões são influenciadas pelos *stakeholders*, organização que desenvolve o software, ambiente técnico e experiência dos responsáveis pela definição da arquitetura.<sup>28</sup>

Na atividade de Projeto, as decisões de projeto tomadas têm reflexos diretos nos atributos de desempenho, confiabilidade e manutenibilidade do software. Se o desempenho for crítico, a arquitetura deve ser projetada de tal forma que haja pouca comunicação dentro do sistema; se a confiabilidade for crítica, a arquitetura deve levar em conta mecanismos de tolerância a defeitos ou falhas e se a manutenibilidade for crítica, a arquitetura deve ser projetada usando componentes de baixa granularidade e autocontidos para facilitar a sua pronta substituição. Por outro lado, os componentes de alta granularidade melhoram o desempenho e os de baixa granularidade melhoram a manutenibilidade. As decisões de projeto devem ser voltadas ao reuso de componentes.<sup>8</sup>

Da mesma forma que a Análise, o Projeto é um processo criativo cujo resultado deve ser uma estrutura do sistema que satisfaça os requisitos funcionais e não funcionais do sistema. Diferentemente da Análise, no Projeto a questão a ser respondida é: “Como o software será desenvolvido?”. A resposta a essa questão é a arquitetura do software, representada por um modelo de arquitetura. Nesta seção são apresentados alguns exemplos de modelos de arquitetura.

Os modelos de arquitetura são usados para documentar as decisões de projeto de arquitetura. Existem modelos estáticos de estrutura, que mostram os componentes principais de sistema, modelos dinâmicos de processo, que

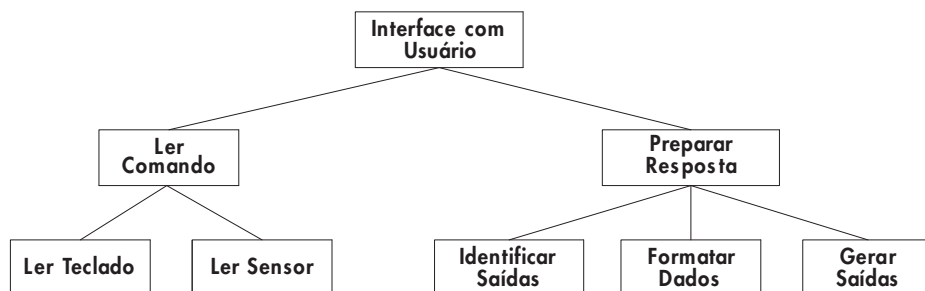
mostram a estrutura do sistema em tempo de execução, modelos de interface, que definem as interfaces de subsistemas, modelos de relacionamentos, que mostram os relacionamentos entre subsistemas e modelos de distribuição, que mostram como subsistemas são distribuídos pelos computadores.<sup>8</sup>

Para orientar as decisões de projeto existem estruturas ou estilos de arquitetura de sistemas mais usados atualmente que são: o modelo baseado em repositório, modelo cliente-servidor, modelo em camadas ou suas combinações (ver mais detalhes a seguir).<sup>8, 28, 29</sup>

Uma vez definido um modelo de arquitetura do sistema, a próxima decisão é qual abordagem de decomposição do sistema em módulos será aplicada. Um módulo é um componente de sistema que fornece serviços para outros componentes, mas não é normalmente considerado como um sistema separado, como é o caso de um subsistema.

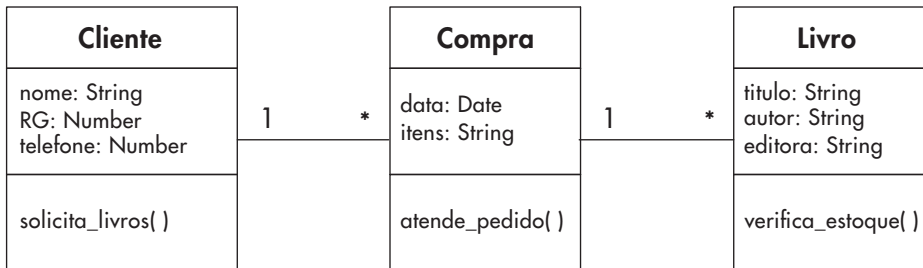
Existem atualmente duas abordagens de decomposição modular mais usadas: na primeira, orientada a funções, o sistema é decomposto em módulos funcionais organizados hierarquicamente e na segunda, Orientada a Objetos, o sistema é decomposto em objetos dispostos em um plano único (não hierarquizados).

O modelo de arquitetura orientado a funções é derivado do Diagrama de Fluxo de Dados (DFD) definido na atividade de Análise. A forma básica do modelo é a transformação dos dados de entrada em dados de saída. A representação deste modelo é conhecida como Diagrama de Estrutura de Módulos (DEM). A Figura 4.9 apresenta um Diagrama de Estrutura de Módulos de um Sistema de Interface com o Usuário.



**Figura 4.9** – Diagrama de Estrutura de Módulos de um Sistema de Interface com o Usuário.

O modelo de arquitetura orientado a objetos (representado pelo Diagrama de Classes) estrutura o sistema em um conjunto de classes com interfaces bem definidas e não fortemente acopladas. As classes chamam serviços oferecidos por outras classes. O modelo é o detalhamento das classes definidas na atividade de Análise. É necessário também acrescentar classes de infraestrutura para apoiar o funcionamento do sistema, por exemplo, classes de iniciação, de finalização, de tratamento de falhas, entre outros, que não foram consideradas anteriormente. Os atributos e métodos das classes devem ser totalmente especificados. A Figura 4.10 apresenta um modelo orientado a objetos que é o Diagrama de Classes detalhado, visto na atividade de Análise.



**Figura 4.10** – Diagrama de Classes detalhado.

Para especificar completamente os módulos (por exemplo, funções e métodos de classes) definidos, existem técnicas muito usadas nos projetos de software. As técnicas são Tabelas de Decisão, Fluxogramas e Pseudocódigo. Essas técnicas permitem especificar os algoritmos que transformam entradas em saídas do software.

Assim que é a finalizada toda a modelagem do sistema em níveis de detalhes que permitem a sua codificação, um documento consolidado deve ser elaborado. Este documento é a Especificação de Projeto.

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

O ponto importante da atividade de Projeto, antes da codificação do software, é definir os módulos de software e realizar decisões sobre algumas alternativas de estrutura de relacionamentos entre os módulos. Quanto mais organizados forem os módulos, mais fácil será a manutenção do software no futuro. Quanto mais coesos e menos acoplados forem os módulos de software, melhores o desempenho e a manutenibilidade do software.

O conceito de coesão refere-se ao grau de integração de funções que têm objetivos comuns dentro de um módulo. O conceito de acoplamento refere-se ao grau de dependência de um módulo em relação ao outro. Esses conceitos surgiram na abordagem Estruturada, mas podem ser usados também na abordagem Orientada a Objetos.

Embora os conceitos de coesão e acoplamento devam ser atendidos, existem diferenças entre as abordagens na atividade de Projeto. Na abordagem Estruturada, a arquitetura do software é obtida a partir de um diagrama DFD detalhado. O DFD é transformado em uma estrutura hierárquica observando a sequência de tratamento dos dados entre as funções. Essa transformação de um modelo (funções) para outro (módulos relacionados) pode ocasionar erros de interpretação resultando em uma arquitetura inadequada. Na abordagem Orientada a Objetos, a arquitetura do software é o resultado do detalhamento do diagrama de classes complementado por classes de infraestrutura e padrões de projeto.

Um fator importante para as decisões de arquitetura do software é a estrutura ou o estilo de arquitetura de sistemas que será usado na implantação do software.

## EXERCÍCIOS

1. Qual é a importância da atividade de Projeto em um desenvolvimento de software?
2. Quais são as entradas para a atividade de Projeto?

3. Suponha que você é o responsável pela definição da estrutura de arquitetura de um software. Quais critérios você deve seguir para esta definição?
4. Quais são os possíveis erros que alguém pode cometer ao adotar a abordagem orientada a funções para definir um modelo de arquitetura do software?
5. Dê exemplo de um software com alto acoplamento. Como o acoplamento poderia ser reduzido?
6. Cite um software com baixa coesão. Como a coesão poderia ser aumentada?
7. O que significa detalhar os módulos após a definição do modelo de arquitetura do software tanto na abordagem Estruturada como na Orientada a Objetos?
8. A abordagem Orientada a Objetos é melhor do que a Estruturada do ponto de vista de manutenibilidade de software? Justifique.
9. A decomposição modular supõe um tratamento *top-down* do sistema para identificar os módulos de software. Compare a decomposição modular na abordagem Estruturada e Orientada a Objetos.
10. É possível desenvolver a atividade de Projeto na abordagem Orientada a Objetos se a atividade Análise foi feita na abordagem Estruturada? Justifique.

## SUGESTÕES DE LEITURA

BASS, Len; CLEMENTS, Paul; KAZMAN, Rick. *Software architecture in practice*. 2<sup>nd</sup> edition. USA: Addison-Wesley Professional, 2003.

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1994.

GARLAN, David; SHAW, Mary. An introduction to software architecture. CMU-CS-94-166. Carnegie Mellon University, 1994.

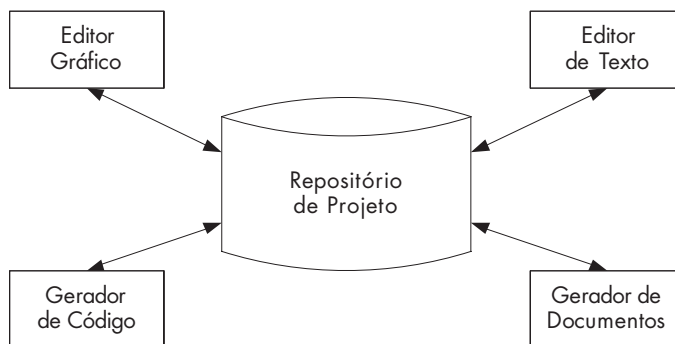
IEEE. *ANSI/IEEE Std1471-2000, recommended practice for architectural description of software-intensive systems*. IEEE, 2000.

MAIER, Mark W.; EMERY, David; HILLIARD, Rich. Software architecture: introducing IEEE standard 1471. *IEEE Computer*, vol. 34, issue 4, pp. 107-109, Apr., 2001.

### 4.3.1. Estilos de Arquitetura de Sistemas

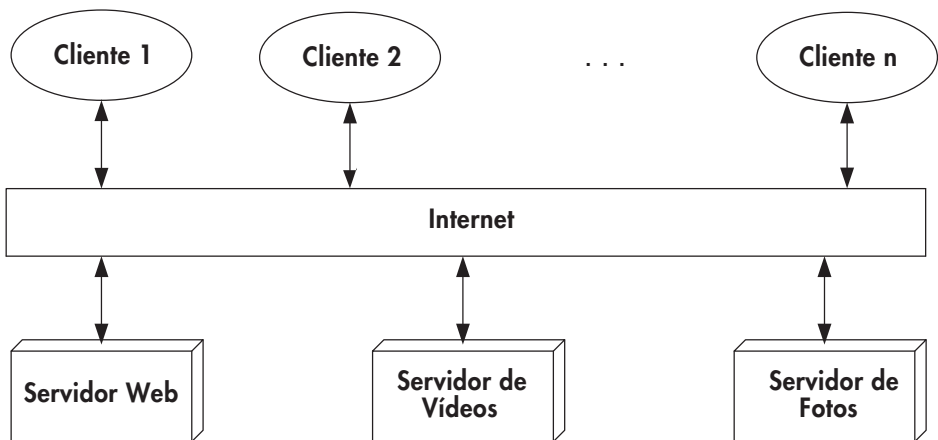
*Modelos de arquitetura são representações do sistema através de subsistemas e suas interfaces. O agrupamento de subsistemas pode ser feito seguindo algumas estruturas ou estilos de arquitetura de sistemas (ou tipos de modelo de arquitetura) mais conhecidos. Um desses estilos é o modelo baseado em repositório onde os subsistemas devem compartilhar grandes quantidades de dados e usam um banco de dados central. Outro é o modelo cliente-servidor onde aparece o conceito de servidores de serviços específicos para clientes interessados nestes serviços. O modelo em camadas permite uma organização dos subsistemas em camadas especializadas que se comunicam por meio de interfaces.*

O modelo baseado em repositório é adequado quando os subsistemas devem compartilhar grande quantidade de dados. Isso pode ser feito de duas maneiras: os dados compartilhados são mantidos em um banco de dados central ou repositório e podem ser acessados por todos os subsistemas; e cada subsistema mantém seu próprio banco de dados e passa dados explicitamente para outros subsistemas. Por exemplo, sistemas de informação, sistemas CAD e ferramentas CASE integradas são bons representantes deste estilo. A Figura 4.11 apresenta um modelo de arquitetura de ferramentas CASE integradas.



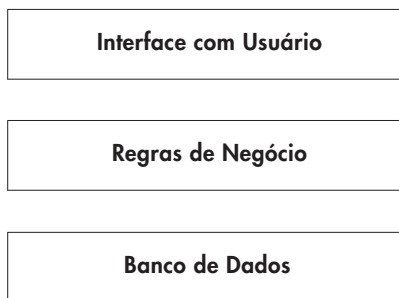
**Figura 4.11** – Modelo baseado em repositório de ferramentas CASE integradas.

O modelo cliente-servidor é adequado quando o sistema é organizado como um conjunto de serviços e servidores/clientes que acessam esses serviços. Os componentes desse modelo são servidores dedicados que fornecem serviços específicos tais como impressão, gerenciamento de dados etc., clientes que chamam estes serviços e uma rede que permite aos clientes acessar os servidores. A Figura 4.12 apresenta um modelo de arquitetura de um Sistema de Acervo de Vídeos e Fotos.



**Figura 4.12** – Modelo cliente-servidor de um Sistema de Acervo de Fotos e Vídeos.

O modelo em camadas é usado para modelar as interfaces de subsistemas. O modelo organiza o sistema em um conjunto de camadas, sendo que cada uma delas fornece um conjunto de serviços. A estrutura em camadas possibilita o desenvolvimento incremental dos subsistemas em camadas diferentes, de tal modo que, quando uma camada de interface muda, somente a camada adjacente é afetada. A Figura 4.13 apresenta um modelo em três camadas de um sistema de informações genérico.



**Figura 4.13** – Modelo em três camadas de um sistema de informações genérico.

Existem diversas outras estruturas ou estilos de arquitetura, alguns gerais e outros para um domínio específico de aplicação.

O **modelo de implantação** é usado para representar a alocação do software em processadores. Serve para mostrar em quais unidades físicas o software estará alocado e quais elementos de comunicação estarão envolvidos.

O **modelo de domínio específico** é usado para representar uma estrutura customizada para uma família de aplicações como sistema de controle de veículos. Essa especialização permite desenvolver padrões, reusar software e aumentar a produtividade de software.

O **modelo de controle** de processos é usado para representar o controle dinâmico de um ambiente físico, comumente representado por um controlador central que gerencia a execução de conjunto de processos associados a sensores e atuadores.

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

A estrutura ou o estilo de arquitetura orienta as decisões de projeto de software.

Podem ocorrer duas situações que devem ser levadas em conta em um projeto de software. Se já existir um estilo de arquitetura, este será uma restrição a ser atendida pelo software. Caso o estilo também seja definido dentro do projeto do sistema, então, existirão mais alternativas de projeto a



serem analisadas para o software. Algumas condições do lado do software podem ajudar a definir o estilo de arquitetura do sistema como um todo, por exemplo, necessidade de segregação de funções críticas.

Em projetos de software mais comuns, os estilos já estão definidos, pois foram usados em projetos de outros sistemas que estão em funcionamento. Também ocorrem situações onde a definição do estilo é uma decisão corporativa, legal e estratégica.

## EXERCÍCIOS

1. Descreva um exemplo de sistema organizado no estilo baseado em repositório. Quais são os dados compartilhados?
2. Descreva um exemplo de um sistema organizado no estilo cliente-servidor. Quais são as funções do cliente e do servidor?
3. Descreva um exemplo de sistema organizado no estilo em três camadas. Identifique tais camadas.
4. Escolha uma aplicação e elabore um modelo de implantação identificando as unidades físicas associadas para cada função de software.
5. Um *framework* de aplicação implementa uma estrutura padrão para o desenvolvimento de uma aplicação em um ambiente específico. A qual estilo de arquitetura está relacionado?
6. Se um sistema é de tempo real, qual dos estilos de arquitetura discutidos é o mais adequado? Justifique.
7. Suponha que um estilo de arquitetura deve ser definido. Sugira critérios que podem ser aplicados para esta definição.
8. Um bom exemplo de ferramentas CASE integradas é o Rational Suite da empresa IBM. Descreva a sua arquitetura relacionando-a ao modelo baseado em repositório.
9. Suponha que um sistema de administração de hotel é projetado e você deve definir o seu estilo de arquitetura. Você deve considerar que futuramente todos os hotéis da rede serão integrados. Qual seria o estilo mais adequado? Justifique.
10. O modelo de visão de arquitetura 4+1 proposto por Kruchten<sup>30</sup> permite descrever a arquitetura de sistemas intensivos de software baseado no uso de múltiplas visões concorrentes. Descreva o modelo de visão 4+1 do ponto de vista dos estilos de arquitetura discutidos.

## SUGESTÕES DE LEITURA

BASS, Len; CLEMENTS, Paul; KAZMAN, Rick. *Software architecture in practice*. 2<sup>nd</sup> edition. USA: Addison-Wesley Professional, 2003.

GARLAN, David; SHAW, Mary. *An introduction to software architecture*. CMU-CS-94-166. Carnegie Mellon University, 1994.

KRUCHTEN, Philippe. Architectural blueprints – the “4+1” view model of software architecture. *IEEE Software*, vol. 12, issue 6, pp. 42-50, Nov., 1995.

PFLEEGER, Shari L. *Software engineering: theory and practice*. 2<sup>nd</sup> edition. New Jersey: Prentice Hall, 2001.

PRESSMAN, Roger S. *Software engineering – a practitioner’s approach*. 6<sup>th</sup> edition. New York: McGraw Hill, 2007.

SOMMERVILLE, Ian. *Engenharia de Software*. Tradução: Kalinka Oliveira e Ivan Bosnic. Revisão Técnica: Kechi Hirama. 9. ed. São Paulo: Prentice Hall, 2011.

## 4.4. CODIFICAÇÃO

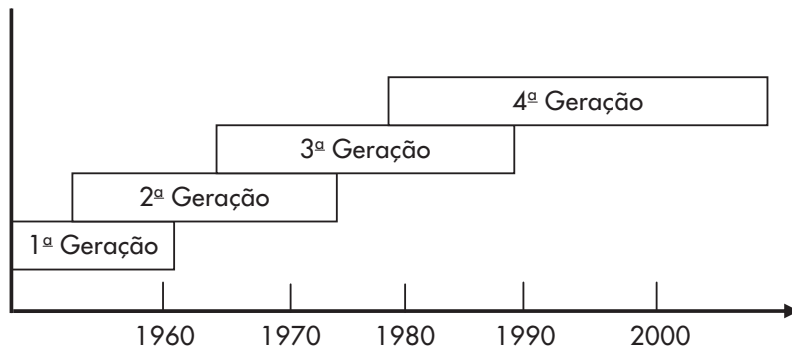
*A atividade de Codificação tem por objetivo traduzir as especificações de software em códigos de programa que possam ser processados por um sistema computacional. Para isto, usam-se linguagens de programação definidas para o projeto. Espera-se que um programa apresente boa qualidade, refletida nas seguintes características: ser modular, funcionar, funcionar de acordo com as especificações, ser flexível, bem estruturado, sem defeitos, ter bom desempenho e ser bem documentado.*

A atividade de Codificação tem por objetivo traduzir as especificações de software em códigos de programa que possam ser processados por um sistema computacional. Naturalmente, esses códigos são dependentes da linguagem de programação escolhida. As características da linguagem e o estilo de codificação podem afetar profundamente a qualidade e a manutenibilidade do software.

Uma linguagem de programação deve ser escolhida segundo alguns critérios, tais como:<sup>26</sup>

- Área de aplicação: industrial, comercial, financeira etc.
- Complexidade algorítmica: múltiplos caminhos de processamento.
- Complexidade de estrutura de dados: filas, listas etc.
- Requisitos de desempenho: tempo de resposta, taxa de transferência de dados (*throughput*) etc.
- Ambiente de execução: plataformas Java, Windows, .NET etc.
- Existência de ferramentas de desenvolvimento: ferramentas CASE.

As várias linguagens de programação atualmente conhecidas acompanharam a evolução dos sistemas computacionais. Assim, as linguagens podem ser classificadas em 1<sup>a</sup>, 2<sup>a</sup>, 3<sup>a</sup> e 4<sup>a</sup> gerações, conforme a Figura 4.14.



**Figura 4.14** – Gerações de linguagens de programação (adaptado).<sup>26</sup>

A 1<sup>a</sup> geração (máquina), que se inicia com o aparecimento do primeiro computador digital (Electrical Numerical Integrator and Computer – Eniac) na década de 1940 e vai até o início da década de 1960, é constituída de código de máquina e linguagem Assembly.

A 2<sup>a</sup> geração vai do fim da década de 1950 ao início da década de 1970 e é constituída, entre outras, das linguagens Fortran, Cobol, Algol e Basic.

A 3<sup>a</sup> geração, de meados da década de 1970 ao fim da década de 1980, é constituída, entre outras, das linguagens Pascal, C, C++, Smalltalk, Lisp e Prolog.

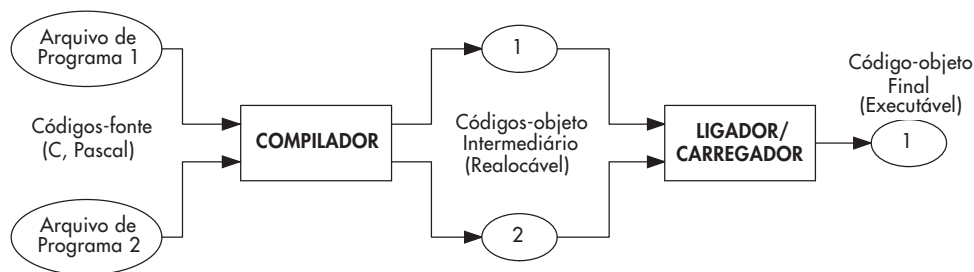
A 4<sup>a</sup> geração, do fim da década de 1970 até os dias de hoje, é constituída entre outras de linguagens de consulta (SQL), geradores de programas

(para Cobol e C), linguagens de prototipação (Visual Basic e Delphi). A característica desta última classe de linguagens é baseada nas técnicas de 4ª geração, em que os desenvolvedores podem descrever os resultados desejados e não o procedimento desejado, numa linguagem não procedimental, diferentemente das linguagens de 3ª geração.

Existem muitas opções de linguagens para desenvolver um programa com qualidade. As qualidades de um bom programa podem ser resumidas em: é modular, funciona, funciona de acordo com as especificações, é flexível, sem defeitos, bem estruturado, tem bom desempenho e é bem documentado.

O documento típico dessa atividade é a Listagem de Programas. Uma listagem de programa, também conhecida como código-fonte, é escrita por um programador em alguma linguagem de programação. O código-fonte deve obedecer às regras de sintaxe da linguagem definida. Além disso, o código deve ser inteligível, favorecendo a clareza e a simplicidade. Deve atender a um estilo de programação.<sup>31</sup>

O processo de geração de código (Figura 4.15), propriamente dito, se inicia com a tradução do código-fonte por um programa compilador, que gera um código-objeto intermediário (realocável – endereçamento relativo de código). Código-objeto é o nome dado ao código que é gerado pelo compilador. Em seguida, este e outros códigos-objeto intermediários são ligados através de um programa ligador/carregador que gera um código-objeto final (executável – endereçamento absoluto de código) que deve ser carregado na máquina-alvo (processador).



**Figura 4.15** – Processo típico de geração de código.

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

A atividade de Codificação é fortemente influenciada por uma boa atividade anterior de Projeto. Os conceitos de coesão e acoplamento farão sentido ao produzir um código-fonte. Também, a escolha de uma linguagem de programação adequada associada à área de aplicação é importante.

Porém, a atividade de Codificação é trabalhosa e, portanto, deve ser facilitada pelo uso de ferramentas e reuso de programas.

Atualmente, existem muitas linguagens e ferramentas disponíveis. Entretanto, tem sido uma atividade pautada no uso exclusivo de ferramentas, descuidando-se dos aspectos de qualidade do código.

Atualmente, as ferramentas CASE (veja mais detalhes no Capítulo 9) ou ambientes de desenvolvimento dirigem o programador a usar padrões de codificação, tornando os códigos mais claros, manuteníveis e com qualidade.

## EXERCÍCIOS

1. Segundo os critérios apresentados, o que é mais importante na escolha de uma linguagem de programação? Justifique.
2. A linguagem Assembly ainda é usada em algumas aplicações. Em que área de aplicação esta linguagem ainda é usada? Dê um exemplo.
3. Examine a linguagem Java e descreva suas características em relação à outra linguagem Orientada a Objetos como, por exemplo, a linguagem C++.
4. Se fosse definir um padrão de código-fonte quais seriam os itens para atender ao critério de inteligibilidade?
5. Como se pode assegurar que os códigos gerados pelo compilador ou ligador/carregador não contenham defeitos?
6. Por que não é suficiente ter somente um bom compilador para a geração de código de programa sem defeitos?
7. Um compilador é um programa que traduz um código-fonte em código-objeto. Quais requisitos devem ser considerados em um bom compilador?

8. Existem relatos de experiências com compiladores que tenham efetivamente produzido códigos errados. Você já passou por essa experiência? Descreva o que ocorreu e como o erro foi corrigido.
9. As ferramentas CASE podem dar maior produtividade à produção de código. Quais são os aspectos de codificação que devem ser considerados ao escolher uma ferramenta CASE?
10. Você provavelmente já trabalhou com códigos de programa gerados por outras pessoas. Em linhas gerais, quais foram os problemas encontrados? Como poderiam ser evitados?

### SUGESTÕES DE LEITURA

KERNIGHAN, Brian W.; PIKE, Rob. *The practice of programming*. USA: Addison-Wesley Professional, 1999.

MISFELDT, Trevor; BUMGARDNER, Gregory; GRAY, Andrew; XIAOPING, Luo. *The elements of C++ Style*. Cambridge University Press, 2004.

VERMEULEN, Allan; AMBLER, Scott W.; BUMGARDNER, Greg; METZ, Eldon; MISFELDT, Trevor; SHUR, Jim. *The elements of Java style*. Cambridge University Press, 2000.

## 4.5. TESTES

*A atividade de Testes tem por objetivo descobrir defeitos no software, considerando aspectos estruturais e lógicos do software. A qualidade de um software está intimamente relacionada com a quantidade de defeitos. Quanto menos defeitos no software, mais qualidade. Para atender aos objetivos de teste a um custo adequado, os conceitos, as estratégias, técnicas e métricas do teste devem ser integradas em um processo de teste definido e controlado. A atividade de Teste deve ser apoiada por um Plano de Testes.*

O termo “teste” vem de testum (Latim) que significa pote de barro usado para ensaios com metais para determinar a sua presença ou medir a massa de seus vários elementos. Assim, colocar algo em teste, do inglês to put to

the test vem desta origem.<sup>32</sup> Ou seja, colocar um software em teste significa verificar a presença de defeitos.

A atividade de Testes tem por objetivo verificar os aspectos estruturais e lógicos do software, bem como os seus aspectos sistêmicos, com o intuito de descobrir defeitos no software. O teste procura descobrir defeitos no software quando seu comportamento não está correto ou não está em conformidade com a sua especificação. Assim, um bom caso de teste é aquele que tem uma elevada probabilidade de revelar um defeito ainda não descoberto.<sup>33</sup>

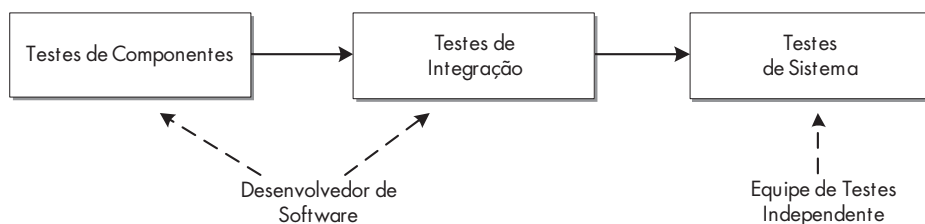
Os desenvolvedores de software reconhecem a importância dos testes. Porém, estes têm sido uma atividade muitas vezes superficial e empírica, além disso, os desenvolvedores, pressionados pelo cronograma do projeto, nem sempre conseguem realizar os testes de forma adequada. Muitas vezes, somente o funcionamento sistêmico é verificado, restando para o cliente continuar os testes no ambiente de produção. Nesse contexto, vale lembrar a frase de Dijkstra:<sup>34</sup>

*O teste de software pode ser usado para mostrar a presença de defeitos, mas nunca a sua ausência.*

Para atender ao objetivo de descobrir defeitos no software a um custo adequado, conceitos, estratégias, técnicas e métricas de teste devem ser integrados em um processo de teste definido e controlado. O processo deve apoiar as atividades de teste e fornecer um guia para as equipes de teste desde o planejamento até a avaliação dos resultados. A Figura 4.16 apresenta as fases de um processo de teste.

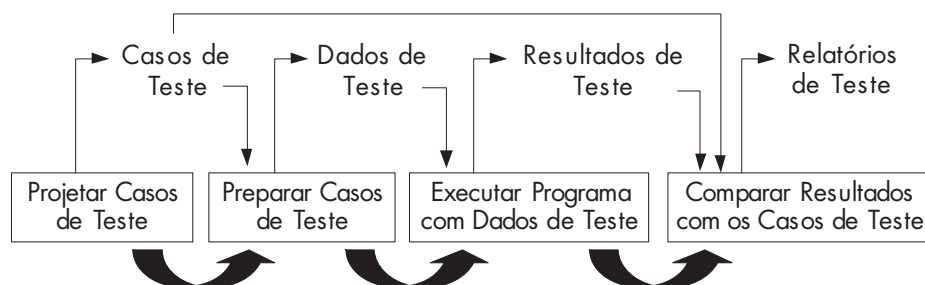
O processo de teste deve ser executado em fases, de forma a torná-lo mais eficaz. A primeira fase, a de Testes de Componentes (ou de Unidades), é realizada pelo próprio desenvolvedor e a segunda, objetiva, testa individualmente os módulos ou componentes de software desenvolvidos – testa-se se o módulo ou componente que realmente executa a tarefa para a qual foi projetado. Em seguida, na fase de Testes de Integração, o desenvolvedor de software deve testar a integração dos componentes do sistema. Por fim, a fase de Testes de Sistema tem por objetivo testar o software do ponto de

vista sistêmico e deve ser realizada por uma equipe independente de testes, com foco no sistema como um todo. Compare esse processo de teste com o Modelo V discutido no Capítulo 2.



**Figura 4.16** – Fases de um processo de teste.

Um procedimento de teste completo tem os seguintes passos:<sup>8</sup> projetar casos de teste, preparar dados de teste, executar programa com dados de teste e comparar resultados com os casos de teste. A Figura 4.17 apresenta esse procedimento.



**Figura 4.17** – Procedimento de teste (adaptado).<sup>8</sup>

O documento típico da atividade de Teste é Plano de Testes. Um Plano de Testes define objetivos para cada tipo (ou fase) de teste, estabelece estratégias de teste, cronograma e responsabilidades, procedimentos e padrões a serem usados na execução e elaboração de relatório de testes, e define critérios para a conclusão do teste, bem como o sucesso de cada teste. A norma IEEE 829-2008<sup>35</sup> descreve o que é necessário para uma boa documentação de teste.



## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

A atividade de Testes é fortemente influenciada pelo seu planejamento antecipado, durante as fases iniciais do desenvolvimento do software. A capacidade de encontrar defeitos está diretamente relacionada a este planejamento.

Embora importantes, os testes são muitas vezes negligenciados por vários motivos, tais como, projeto atrasado, testes não planejados e sem ferramentas adequadas e pessoal qualificado para as atividades. Porém, deve-se descobrir e corrigir o maior número de defeitos possível antes do cliente. Muitos destes problemas podem ser minimizados pelo uso de ferramentas CASE próprios para teste (veja mais detalhes no Capítulo 9).

Outra questão é como determinar o fim dos testes. O fim de um teste pode ser orientado a partir do Plano de Testes. Um Plano de Testes deve definir critérios de conclusão baseados nos resultados obtidos. Se os critérios forem atendidos, considera-se o fim de um teste.

A atividade de Testes é parte dos conceitos de Verificação e Validação (veja mais detalhes no Capítulo 6).

## EXERCÍCIOS

1. Há possibilidade de detectar todos os defeitos de um software? Justifique.
2. Quais são as condições mínimas para realizar um bom teste?
3. A finalização de testes deve obedecer a critérios definidos no Plano de Testes. Cite dois critérios para esta finalização.
4. Quais são os pontos fracos de testes que visam somente o aspecto funcional sistêmico do software?
5. Um teste bem-sucedido é aquele que revela um defeito ainda não descoberto.<sup>33</sup> Quantos testes seriam necessários para descobrir a maioria dos defeitos?
6. Qual deve ser o percentual de esforço alocado aos testes de software? É dependente do tipo de processo de desenvolvimento de software (sequencial ou evolutivo)?

7. Os testes de componentes estão relacionados com a atividade de Codificação. Espera-se que o programador faça os devidos testes nos componentes que desenvolveu. Quais são os problemas desta abordagem?
8. Quais são as diferenças entre teste de integração e teste de sistema?
9. Existem duas estratégias para os testes de integração: *top-down* e *bottom-up*. Pesquise o que são estas estratégias e descreva as suas principais diferenças.
10. Por que os testes de sistema devem ser realizados por uma equipe independente?

## SUGESTÕES DE LEITURA

ADRION, W. Richards; BRANSTED, Martha A.; CHERNIAVSKY, John. C. Validation, verification and testing of computer software. *ACM Computing Surveys*, vol. 14, issue 2, pp. 159-192, Jun., 1982.

BINDER, Robert V. Design for testability in object-oriented systems. *Communications of the ACM*, vol. 87, issue 9, pp. 87-101, Sep., 1994.

DEMILLO, Richard A.; LIPTON, Richard J.; SAYWARD, Frederick G. Hints on test data selection: help for the practicing programmer. *IEEE Computer*, vol. 11, issue 4, pp. 34-41 Apr., 1978.

FRANKL, Phyllis G.; WEYUKER, Elaine J. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, vol. 14, issue 10, pp. 1483-1498, Oct., 1988.

GOODENOUGH, John. B; GERHART, Susan. L. Toward a theory of test data selection. *Proceedings of the International Conference on Reliable Software*, vol. 10, issue 6, pp. 493-510, Jun., 1975.

HONG, Zhu; HALL, Patrick. A. V.; MAY, John. H. R. Software unit test coverage and adequacy. *ACM Computer Surveys*, vol. 29, issue 4, pp. 365-427, Dec., 1997.

NTAFOS, Simeon C. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, vol. 14, issue 6, pp. 868-874, Jun., 1988.

ROYER, Thomas C. *Software testing management. Life on the critical path*. Englewood Cliffs: Prentice Hall, 1993.

WHITTAKER, James A. What is software testing? And why is it so hard? *IEEE Software*, vol. 17, issue 1, pp. 70-79, Jan./Feb., 2000.

## Desenvolvimento de IHM

---

**A**lém das atividades normais de desenvolvimento, existem outras atividades importantes para a produção de um software. Uma delas é o desenvolvimento de interfaces com o usuário, chamada Interface Homem-Máquina (IHM). Atualmente, grande parte das aplicações é voltada para prestação de serviços a vários perfis de usuários. Assim, produzir uma interface amigável com o usuário é um desafio.

---

### 5.1. INTERFACE COM O USUÁRIO

*Atualmente, muitas aplicações são voltadas para usuários que dialogam e obtêm informações dos sistemas computacionais. Uma interface com o usuário permite a comunicação entre o usuário e o sistema. Assim, o projeto de interface exige conhecimentos de fatores humanos e tecnologia de interfaces. O projeto de interface deve levar em conta as necessidades, experiências e capacidades dos usuários de sistema e os projetistas devem estar cientes das limitações físicas e mentais das pessoas (por exemplo, memória limitada de curto prazo) e reconhecer que pessoas cometem erros. Durante o projeto de interface com o usuário, é importante considerar alguns aspectos que vão influenciar diretamente no uso da interface. Estes aspectos estão relacionados ao tempo de resposta, facilidade de ajuda (help online), indicações de mensagens de erro e sequência de comandos.*

Muitas aplicações hoje em dia usam uma interface com o usuário para dialogar e obter informações dos sistemas computacionais. Diferentemente de outras partes dos sistemas, o projeto de interface exige conhecimentos de fatores humanos e tecnologia de interfaces.

As interfaces com o usuário devem ser projetadas para atender às habilidades, experiências e expectativas dos seus usuários. Frequentemente, os usuários de sistemas julgam um sistema pela sua interface ao invés de sua funcionalidade. Um projeto fraco de interface com o usuário é a razão pela qual os usuários cometem muitos erros e muitos sistemas de software nunca são usados.<sup>36</sup>

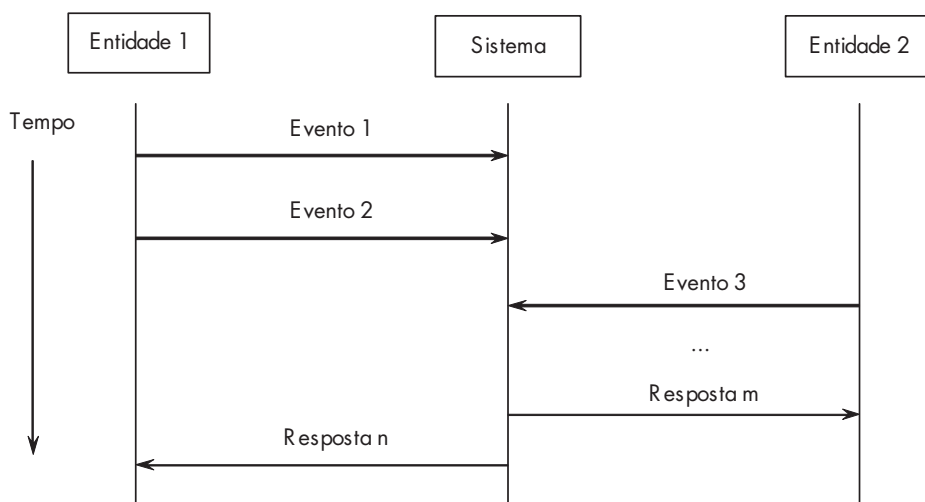
O projeto de interface deve levar em conta as necessidades, experiências e capacidades dos usuários do sistema. Os projetistas devem estar cientes das limitações físicas e mentais das pessoas (por exemplo, memória limitada de curto prazo) e reconhecer que pessoas cometem erros.<sup>36</sup>

Para obter um bom projeto de interface, alguns princípios devem ser seguidos:<sup>36</sup>

- Familiaridade de usuário: A interface deve ser baseada em termos e conceitos de Orientação a Objetos e não de conceitos de computador. Por exemplo, um sistema de escritório deve usar conceitos como cartas, documentos, folhetos ao invés de diretórios e identificadores de arquivos.
- Consistência de interface: O sistema deve apresentar um nível de consistência apropriado. Comandos e menus devem ter o mesmo formato, pontuações de comandos devem ser similares etc.
- Surpresa mínima: Se um comando opera em modo conhecido, o usuário deve ser capaz de prever a operação de comandos similares.
- Facilidade de recuperação: O sistema deve fornecer alguma resistência a erros dos usuários e permitir que eles sejam corrigidos. Isto poderia incluir um recurso de "refazer", confirmação de ações destrutivas, deletes etc.
- Guia de usuário: Alguns guias de usuários como sistemas de *help*, manuais *online* etc. devem ser fornecidos.
- Diversidade de usuário: Recursos de interação para tipos diferentes de usuários devem ser apoiados. Por exemplo, alguns usuários têm dificuldades de visão e, assim, textos maiores devem ser disponíveis.

Uma das formas de iniciar o projeto é desenvolver modelos que representem fielmente a interação que será realizada por um usuário. O modelo de usuário descreve o perfil dos usuários finais do sistema. Levam-se em conta os tipos de usuários desde principiantes (aqueles que não são familiarizados com informática), instruídos e intermitentes (aqueles que usam informática esporadicamente, mas dominam o assunto) até instruídos e frequentes (aqueles que usam informática frequentemente e dominam o assunto). O modelo de tarefa é usado para entender as tarefas que as pessoas realizam atualmente de modo que se possa derivar um conjunto de tarefas que se acomodem no modelo de usuário.<sup>3</sup> Uma das formas práticas de entender essas tarefas é por meio de etnografia, que é uma técnica de observação do usuário nas atividades de seu dia a dia no trabalho: ele observa suas interações com os outros e o uso de recursos do ambiente.<sup>36</sup>

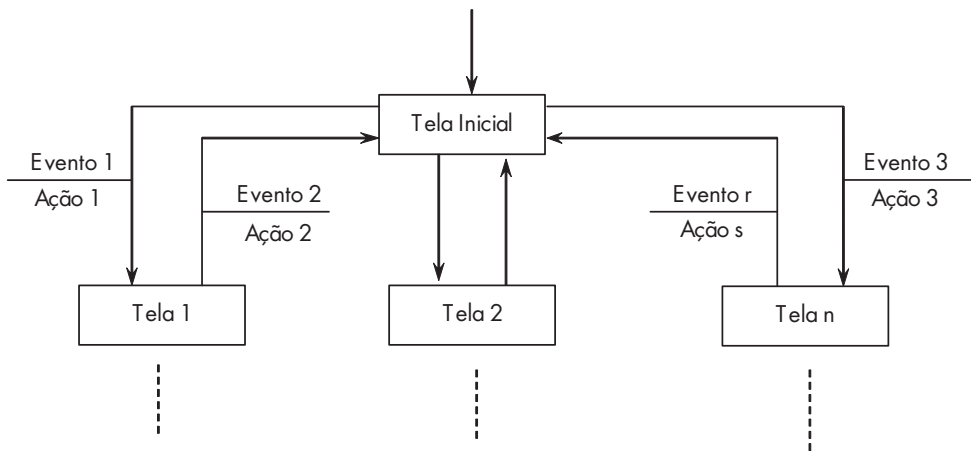
Uma modelagem de interfaces pode ser feita por meio de cenários. Um cenário é uma sequência de eventos/respostas que fluem entre as entidades do ambiente e o sistema, de acordo com uma finalidade. A Figura 5.1 apresenta um exemplo de modelo de interface através de cenário.



**Figura 5.1** – Modelo de interface por meio de um cenário.

Neste exemplo, um cenário poderia ser a reserva de quarto em hotel. A entidade 1 sendo um cliente e a entidade 2 um atendente. O sistema seria um sistema de gerenciamento de hotel. Os eventos correspondem ao pedido, à verificação de vaga e a confirmação de reserva.

Modelos de navegação são outra forma de modelagem. Um modelo de navegação é baseado em um diagrama de transição de estados,<sup>20</sup> no qual os estados são representados pelo tipo de tela e as transições são representadas por eventos e ações que ativam as funções da aplicação de software. A Figura 5.2 apresenta um modelo de navegação representado por um diagrama de transição de estados, ou mais claramente, um diagrama de transição de “telas”.



**Figura 5.2** – Modelo de navegação por diagrama de transição de estados.

Durante o projeto de interface com o usuário é importante considerar alguns aspectos que vão influenciar diretamente no uso da interface. Estes aspectos estão relacionados ao tempo de resposta, facilidade de ajuda (*help online*), indicações de mensagens de erro e sequência de comandos.<sup>36</sup>

O tempo de resposta é função da sua duração e variabilidade. A duração pode ser longa ou curta e a variabilidade está associada ao desvio médio de tempo de resposta. Uma pequena variabilidade permite estabelecer uma interação mais harmônica entre o usuário e o sistema. Uma facilidade de

ajuda (*help online*) permite que o usuário faça consultas e tire dúvidas sobre a interface sem que ele precise abandoná-la. As indicações de mensagens devem ser de fácil entendimento, oferecer conselhos, avisar sobre consequências negativas no caso de erros cometidos e, se possível, vir acompanhadas de um sinal visual ou sonoro.

Atualmente, existem vários ambientes de desenvolvimento de interfaces para apoiar o projeto e a implementação de interfaces. Estes ambientes oferecem ferramentas que facilitam a criação de janelas, interação de dispositivos, mensagem de erros, comando e muitos outros elementos de um ambiente interativo. Exemplos de ambientes são Delphi, Visual Basic e Eclipse.

Para desenvolver uma interface com o usuário é importante também observar algumas normas que podem orientar a adoção de melhores práticas de projeto de interface. A norma NBR 9241-11 define a usabilidade e explica como identificar a informação necessária a ser considerada na especificação e avaliação de usabilidade de um computador em termos de medidas de desempenho e de satisfação do usuário.<sup>37</sup> A usabilidade é uma medida na qual um produto pode ser usado por usuários definidos para atingir seus objetivos com eficácia, eficiência e satisfação em um contexto específico de uso.

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

Existem muitas ferramentas e ambientes disponíveis para o desenvolvimento de interfaces com o usuário. É sempre bom lembrar que uma interface não é apenas estética, mas uma porta de entrada do usuário aos serviços disponibilizados. Assim, o convite ao usuário deve ser feito por uma interface que o ajude a chegar de forma rápida e organizada à informação desejada ou realizar um serviço de maneira segura e confiável.

Dadas as características dos sistemas atuais, notadamente com serviços disponibilizados via internet, pode-se considerar a interface com o usuário um projeto à parte. Muitas vezes, o sucesso de um projeto de software deve-se a uma boa interface. Pode-se dizer que o inverso é também verdadeiro.

## EXERCÍCIOS

1. Descreva com suas palavras o que é usabilidade.
2. Apoiando-se em algum sistema que você use regularmente, os princípios de projeto de interface com o usuário descritos foram seguidos?
3. Os projetos de software que você tem participado usaram conceitos de usabilidade?
4. As tarefas dos usuários são normalmente consideradas nos seus projetos de interface? De que forma são analisadas?
5. Elabore um modelo de cenário para terminais de caixa-eletrônico.
6. Escolha uma loja de comércio eletrônico e elabore um modelo de navegação de suas páginas.
7. Existem algumas técnicas para avaliar o desempenho de uma interface com o usuário. Uma delas é embutir um software monitor na interface e analisar a frequência de uso dos recursos disponíveis. Como isso pode ser abordado com os usuários?
8. Determinadas aplicações devem levar em conta a diversidade de usuários. Sugira uma forma de considerar esta diversidade no projeto da interface com o usuário.
9. Considere a facilidade de ajuda (*help online*) ao usuário durante a operação de uma interface com o usuário. Faça uma lista de recursos, em ordem de prioridade, que devem ser oferecidos a ele.
10. A etnografia pode ser usada para analisar as tarefas das pessoas no ambiente de trabalho e influenciar as decisões de projeto de interface. Observe uma pessoa que usa um aplicativo e identifique duas situações que podem ser consideradas no projeto de uma nova interface com o usuário.



## SUGESTÕES DE LEITURA

ABNT. NBR 9241-11. *Requisitos ergonômicos para trabalho de escritórios com computadores – Parte 11: orientações sobre usabilidade*. ABNT, 2002.

LEE, Geoff. *Object-oriented GUI application development*. Englewood Cliffs: Prentice Hall, 1993.

NIELSEN, Jakob. *Usability engineering*. San Francisco: Morgan Kaufmann, 1993.

RUBIN, Jeffrey; CHISNELL, Dana. *Handbook of usability testing: how to plan, design, and conduct effective tests*. 2<sup>nd</sup> edition. New York: Wiley, 2008.

## Verificação e Validação

---

Nos capítulos anteriores, a atividade de Testes foi apresentada. Também foi visto que os testes dependem de um código de programa pronto para verificar se ele funciona. No entanto, isso traz algumas questões importantes. E se não funcionar? Quais seriam as causas? Seria suficiente apenas funcionar? Será que, no fim das contas, as expectativas do cliente estariam atendidas?

Verificação e Validação (V&V) é um processo aplicado para melhorar a qualidade dos produtos e a produtividade dos processos de software. V&V permite que os desenvolvedores identifiquem problemas de software o mais rápido possível e os corrijam antes de serem liberados aos usuários. Além disso, permite identificar e corrigir problemas nas atividades de desenvolvimento para aumentar a produtividade de novos projetos de software.

---

### 6.1. CONCEITOS DE VERIFICAÇÃO E VALIDAÇÃO

*Verificação e Validação (V&V) permite que se determine sistematicamente se os requisitos de um sistema estão sendo corretamente tratados e implementados. V&V é um processo que diminui a chance de retrabalho desde as primeiras fases do desenvolvimento de software. V&V contribui diretamente para o atendimento dos prazos e custos do projeto. Portanto, conhecer os conceitos de V&V é importante para obter produtos de maior qualidade e produtividade e melhorar os processos continuamente.*

Verificação e Validação é o nome dado ao processo para determinar se os requisitos de um sistema ou componente estão completos e corretos, os produtos de cada fase de desenvolvimento atende aos requisitos e às condições impostas pela fase anterior e o sistema ou componente final está aderente com os requisitos especificados.<sup>2</sup>

Requisitos incompletos e incorretos indicam problemas no produto que podem se manifestar por meio de falhas, caso não sejam corrigidos, durante a operação e uso do software. Antes de propor formas de eliminação de tais problemas, cabe ressaltar algumas definições importantes, segundo o IEEE:<sup>2</sup>

- a) Erro (*error, mistake*): Uma ação humana que produz um resultado incorreto.
  - Os desenvolvedores cometem erros (enganos) quando interpretam mal as necessidades dos clientes.
  - Os usuários cometem erros (enganos) quando operam um sistema em desacordo com as intenções dos desenvolvedores.
- b) Defeito (*bug, fault, defect*): Implementado dentro de um artefato.
  - Requisitos inconsistentes com as necessidades dos clientes.
  - Requisitos funcionais do sistema em desacordo com os requisitos de negócio.
- c) Falha (*failure*): A incapacidade de um sistema ou componente em executar as funções requeridas dentro de um nível de desempenho requerido.
  - Manifestação de um defeito no sistema ou software.
  - Apresentação de datas incorretas.
  - Tela azul no monitor do computador.

A partir dessas definições, pode-se dizer que erros causam defeitos. Os defeitos podem ou não se manifestar em falhas. Entretanto, existe a possibilidade de se cometer erros em qualquer fase de desenvolvimento do software. Assim, o processo de V&V deve ser aplicado em todo o ciclo de desenvolvimento.

A verificação do V&V é o processo de avaliação de um sistema ou componente para determinar se os produtos de uma dada fase de desenvolvimento satisfazem às condições impostas para o início desta fase (“Estamos construindo certo o produto?”). Já a validação do V&V é o processo de

avaliação de um sistema ou componente durante ou no fim do processo de desenvolvimento para determinar se ele satisfaz aos requisitos especificados (“Estamos construindo o produto certo?”).<sup>2</sup>

Existem técnicas de V&V que podem ser divididas em estáticas e dinâmicas.<sup>36</sup>

Nas técnicas estáticas, a avaliação de um produto de software é realizada por um grupo de revisores, com intuito de identificar defeitos. Podem ser:

- a) **Revisões Técnicas e *Walkthroughs*:** Indicadas para todas as fases do ciclo de desenvolvimento de software.<sup>38</sup>
  - Revisões Técnicas permitem avaliar um produto de software para determinar a sua adequação ao uso pretendido. O objetivo é identificar discrepâncias entre especificações e padrões aprovados.<sup>7</sup>
  - *Walkthroughs* permitem também avaliar um produto de software. O objetivo é identificar anomalias, melhorar o produto de software, considerar alternativas de implementação e avaliar conformidade a padrões e especificações.<sup>7</sup>
- b) **Inspecções:** Indicadas para a fase de codificação.<sup>38</sup>
  - Inspecções têm por objetivo detectar e identificar anomalias no produto de software.

Essas técnicas são conhecidas atualmente como Inspecções de Software aplicáveis em todas as fases do ciclo de desenvolvimento de software. Portanto, os defeitos podem ocorrer em códigos de programas, arquitetura, requisitos, especificações e documentação em geral.

Durante um processo de inspeção (veja mais detalhes no Capítulo 6) alguns defeitos podem ser identificados. Os tipos de defeitos podem ser caracterizados como:

- **Incorreção:** Implementação incorreta da especificação do cliente/usuário.
- **Ausência:** Checagem de requisito especificado que não foi incorporado no produto.
- **Extra:** Checagem de requisito não especificado e incorporado no produto.

Outra forma de ver os defeitos é pelo tipo de danos que eles podem causar no software. Os defeitos então podem ser:

- Pequenos: Podem ser corrigidos rapidamente e não causam mau funcionamento do software. Ex: defeitos de digitação, omissões em textos que precisam ser esclarecidos para seu entendimento etc.
- Grandes: São defeitos relativos às especificações que podem causar mau funcionamento do software. Ex: ausência de funções, problemas de interfaces etc.
- Muito sérios: São defeitos que podem comprometer o projeto ocasionando o reprojeito total (ou quase) e a recodificação do software.

Se os defeitos podem ser embutidos em documentação de software em geral, eles precisam ser identificados, descritos e contados, além de corrigidos. Dessa maneira, podem-se melhorar as atividades de desenvolvimento e a qualidade e produtividade do software.

Nas técnicas dinâmicas, a avaliação de um produto de software é feita através de testes. Os capítulos anteriores já apresentaram os testes em processos de software e como uma atividade é importante no ciclo de desenvolvimento de software. Aqui, apresentam-se mais detalhes para complementar estes capítulos.

Os testes podem ser estruturais e funcionais. As técnicas usadas para esses testes são conhecidas como teste Caixa Branca (estrutural) e teste Caixa Preta (funcional).

Para realizar os primeiros testes, a boa prática diz que se deve verificar inicialmente se a menor unidade de software está funcionando de acordo com as suas especificações. Nestes realizam-se os testes estruturais. Após a verificação das unidades, parte-se para os testes funcionais para verificar o software como um todo.

Os testes Caixa Branca atendem às seguintes características:

- O projeto de casos de teste usa a estrutura de controle procedimental do software (fluxo de controle do software) para derivar casos de teste.
- Deve garantir que todos os caminhos independentes dentro de um módulo tenham sido exercitados pelo menos uma vez.

- Deve exercitar todas as decisões lógicas para valores falsos ou verdadeiros.
- Deve executar todos os laços em suas fronteiras e dentro de limites operacionais.
- Deve exercitar as estruturas de dados internas para garantir a sua validade.

Os testes Caixa Preta atendem às seguintes características:

- Concentram-se nos requisitos funcionais do software.
- São uma abordagem complementar aos testes estruturais.

Os tipos de teste mais comuns realizados para cobrir os testes estruturais e funcionais são:

- Teste de unidade (estrutural): O objetivo é testar os módulos isoladamente verificando o funcionamento conjunto dos algoritmos e as estruturas de dados. A referência usada para os testes é a Especificação de Módulos, um documento detalhado de cada módulo de software. O teste é realizado normalmente pelo programador da unidade ou módulo de software.
- Testes de integração (funcional): O objetivo é testar um conjunto de módulos verificando o seu funcionamento com foco nas suas interfaces entre os módulos. A referência usada para os testes é a Especificação de Projeto, um documento detalhado da organização e interdependência entre os módulos de software. O teste é realizado normalmente pela equipe de desenvolvimento do software.
- Testes de validação (funcional): O objetivo é testar o software como um todo verificando se todas as exigências funcionais, comportamentais e de desempenho foram atendidas. A referência usada para os testes é a Especificação de Requisitos de Software, um documento que contém os requisitos funcionais e não funcionais do software. O teste é realizado normalmente também pela equipe de desenvolvimento de software.
- Testes de sistema (funcional): O objetivo do teste é medir o sistema em diferentes cenários verificando se todos os elementos do sistema (hardware, software, banco de dados e pessoas) foram adequadamente integrados e realizam as funções requeridas. A referência usada para os testes é a Especificação de Sistema, um documento que contém a descrição funcional do sistema, dos subsistemas e do seu desempenho no ambiente operacional. O teste é realizado normalmente por uma equipe independente ou um usuário do sistema.

Outros testes de sistema conhecidos são:

- Teste de recuperação: Teste que força o sistema a apresentar falhas de diversas maneiras e verifica se a recuperação (reiniciação do sistema e recuperação de dados) é adequadamente executada.
- Teste de proteção: Teste que tenta verificar se todos os mecanismos de proteção embutidos em um sistema funcionam contra acessos indevidos.
- Teste de estresse: Teste para confrontar o software com situações anormais (alta exigência de recursos, alto número de interrupções, alta taxa de entrada de dados, alta busca de dados em disco etc.).
- Teste de desempenho: Teste do software no contexto de um sistema integrado (tempos envolvidos, ciclos de processador, interrupções etc.).

Existem outras denominações de teste como Teste Alfa (testes realizados pelo cliente no ambiente de desenvolvimento do software) e Teste Beta (testes realizados em um ou mais ambientes do cliente pelos usuários).<sup>3</sup>

Apesar da importância dos testes no desenvolvimento de software, sua realização apresenta algumas dificuldades. Uma delas é a falta de conhecimento necessário sobre testes por parte dos desenvolvedores de software. Outra dificuldade é sua simplificação quando o cronograma do projeto está comprometido.

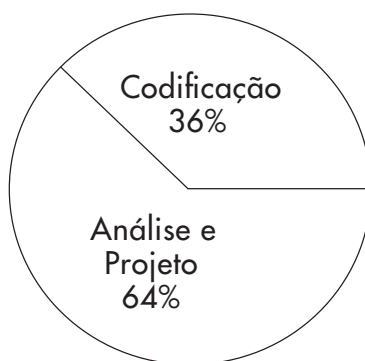
Para isto, alguns princípios importantes podem ser seguidos:<sup>32</sup>

- Teste completo não é possível, ou seja, testar todas as situações não é possível. Não significa que se pode deixar algum requisito do cliente sem teste.
- A atividade de teste é criativa e difícil, ou seja, ela exige boa experiência dos testadores. É necessário conhecimento para ter maior cobertura possível dos testes.
- Uma importante razão do teste é a prevenção de defeitos, ou seja, o teste permite melhorar a qualidade do software detectando os defeitos no software.
- O teste é baseado em risco, ou seja, o esforço de teste é proporcional ao risco de negócio envolvido (resultados incorretos, transação não autorizada, perda de desempenho, comprometimento de segurança etc.). Quanto maior o risco de negócio mais testes devem ser realizados.
- Ele deve ser planejado por se tratar de uma atividade importante que exige estratégia e recursos.

- O teste requer independência, ou seja, não basta planejá-lo, é necessário ter visão crítica para analisar os resultados; uma boa prática é “quem implementa não testa”.

Do ponto de vista de qualidade de software, o teste ajuda a tornar a qualidade visível. Nesse sentido, ele é a maneira de medir a qualidade de software.<sup>32</sup>

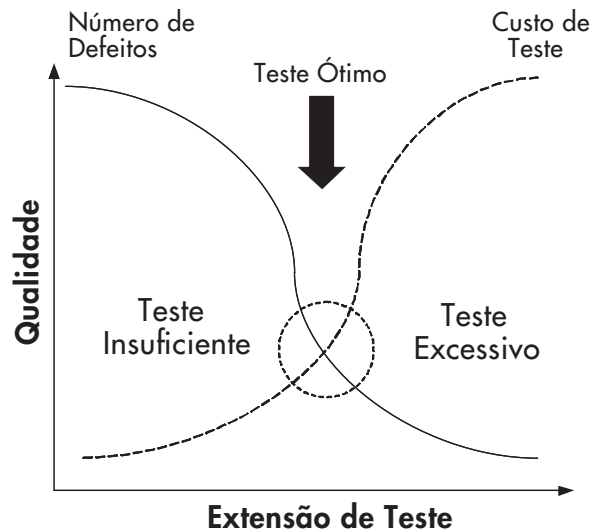
Os defeitos são embutidos nas fases anteriores aos testes. Portanto, deve-se corrigir o maior número de defeitos antes da atividade de Testes. Segundo Perry,<sup>39</sup> a proporção de defeitos detectados em projetos de software pode ser vista na Figura 6.1.



**Figura 6.1** – Proporção de defeitos detectados em projetos de software.<sup>39</sup>

A qualidade de um software está intimamente relacionada à quantidade de defeitos descobertos. Quanto mais defeitos são descobertos, maior é a qualidade do software. Portanto, pode-se imaginar que, se a atividade de teste fosse realizada durante um longo tempo, a quantidade de defeitos restantes seria a menor possível. No entanto, quanto mais longo o período de testes, mais caro fica o software. A Figura 6.2 apresenta a relação entre qualidade de software e extensão de teste sob o ponto de vista de custo, destacando a região de intersecção representada pelo Teste Ótimo, qualidade com custo adequado.





**Figura 6.2** – Relação entre qualidade de software e extensão de teste.<sup>39</sup>

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

Os conceitos de V&V são muito importantes para a melhoria da qualidade de software. Porém, o processo de V&V, considerando-se as técnicas estáticas e dinâmicas, pode se tornar muito dispendioso, devido aos custos com alocação de recursos e atividades envolvidas. Entretanto, a falta de um processo de V&V pode comprometer o sucesso do desenvolvimento de software. Por isso, o gerente de projetos deve gerenciar as atividades de V&V de modo a produzir resultados efetivos.

Quanto às técnicas estáticas são poucos os exemplos de aplicação na indústria de software, exceto em aplicações críticas. Isso é devido à falta de conhecimento dos benefícios destas técnicas e à escassez de dados para tomar decisões.

Quanto às técnicas dinâmicas, embora sejam mais conhecidas em comparação às técnicas estáticas, elas não são plenamente aplicadas devido,

principalmente, às pressões de cronograma de projeto, que muitas vezes já está comprometido, quando os testes vão ser iniciados. No Capítulo 2, os processos discutidos apresentavam uma atividade de testes prevista. Cada uma delas com uma abordagem própria para verificar a qualidade do software.

Outra atividade importante de teste é a depuração de software realizada pós-teste. A depuração de software é complementar ao teste bem-sucedido, ou seja, ela é realizada após um defeito descoberto.<sup>40</sup> A depuração de software envolve a localização e a correção de defeitos.

## EXERCÍCIOS

1. Em uma verificação de produtos ou documentos buscam-se erros, defeitos ou falhas? Justifique.
2. A maior parte dos defeitos é embutida nas fases iniciais do desenvolvimento de software. Sugira formas de minimizar este problema.
3. Explique com suas palavras o que significam as questões “Estamos construindo certo o produto?” para Verificação e “Estamos construindo o produto certo?” para Validação.
4. Escolha três produtos de desenvolvimento de software que devem passar por uma técnica estática de V&V. Qual foi o critério usado para as escolhas?
5. A aplicação da técnica dinâmica de V&V, conhecido como teste de software, depende dos programas estarem prontos. Como se poderia organizar os testes de maneira que sejam os mais efetivos possíveis?
6. Qual é a diferença entre técnicas estáticas e dinâmicas de V&V? O que podem ser verificados?
7. O teste é baseado em risco ao negócio. Cite uma área de negócio onde os testes devem ser mais rigorosos. Justifique.
8. O que são testes caixa branca e caixa preta? Dê exemplos.
9. O que são testes de integração? Dê um exemplo.
10. A finalidade dos testes de validação é verificar o atendimento de requisitos do cliente. Está correto? Justifique.

## SUGESTÕES DE LEITURA

DELAMARO, Márcio. E; MALDONADO, José C.; JINO, Mario. *Introdução ao teste de software*. Rio de Janeiro: Campus, 2007.

IEEE. *ANSI/IEEE Std 829-2008 – IEEE standard for software and system test documentation*. IEEE, Jul., 2008.

IEEE. *ANSI/IEEE Std 610.12-1990 - IEEE standard glossary of software engineering terminology*. IEEE Standards Collection. Software Engineering. New York: IEEE, 1990.

IEEE. *ANSI/IEEE Std 1028-1988 – IEEE standard for software reviews and audits*. New York: IEEE, 1989.

FAGAN, Michael E. Advances in software inspection. *IEEE Transactions on Software Engineering*, vol. 12, issue 7, pp. 744-751, Jul., 1986.

HETZEL, Bill. *The complete guide to software testing*. 2<sup>nd</sup> edition. Chichester: Wiley, 1993.

PERRY, William E. *Effective methods for software testing*. New York: Wiley, 1995.

ROYER, Thomas. C. *Software testing management. Life on the critical path*. Englewood Cliffs: Prentice Hall, 1993.

WHEELER, David A.; BRYKCZYNSKI, Bill; MEESON Jr., Reginald. N. *Software inspection –an industry best practice*. USA: IEEE Computer Society Press, 1996.

## 6.2. PROCESSO DE INSPEÇÃO

A atividade de Processo de Inspeção foi inspirada no processo de inspeção da indústria fabril. Diferentemente de produtos manufaturados, onde uma inspeção pode determinar a inviabilidade de um lote inteiro, no caso do software, o processo de inspeção é realizado para cada programa individual desenvolvido. Enquanto no primeiro caso, existem padrões e normas para serem atendidas, no caso do software, a norma é baseada no atendimento dos requisitos especificados. O processo de inspeção de software inicialmente dava ênfase no programa (código-fonte) e logo se expandiu para qualquer produto de software obtido ao longo das fases de desenvolvimento de software.

O conceito de inspeção de software ficou conhecido como inspeção de programa (código-fonte) em um trabalho da empresa IBM nos Estados Unidos em 1972.<sup>41</sup> A ideia principal é garantir que o produto de software esteja completo e atenda aos requisitos especificados. Em geral, considera-se que qualquer representação ou documento é passível de ser verificado por uma inspeção de software. Para ter os benefícios da inspeção de software, ela deve ocorrer logo nas fases iniciais de desenvolvimento.

Segundo Wheeler, Brykczynski e Meeson Jr.,<sup>42</sup> isso melhora a produtividade e qualidade do software. A Figura 6.3 apresenta um perfil de custos de desenvolvimento de projetos com inspeção e sem inspeção de software. No caso de desenvolvimentos com inspeção há uma elevação de 15% dos custos no início em relação ao desenvolvimento sem inspeção, porém é 25-35% mais produtivo, pois a curva cai rapidamente em todo o desenvolvimento. No caso de desenvolvimento sem inspeção, os custos iniciais são menores, mas há 44% mais retrabalho, pois as atividades ficam mais concentradas nos testes do que no desenvolvimento com inspeção. O custo de desenvolvimento, representado pela área formada por cada um das curvas, com inspeção é 30% menor do que sem inspeção.

Além da melhoria na qualidade e na produtividade do software, o processo de inspeção aumenta a satisfação do cliente, melhora os prazos de entrega, reduz os defeitos reportados pelos usuários, reduz os erros cometidos pelos desenvolvedores e proporciona melhoria contínua do próprio processo de inspeção pela remoção sistemática de defeitos.

Para ser eficaz, a inspeção de software deve ser realizada através de um conjunto de atividades organizadas em um processo. Esse processo pode ser visto através dos papéis envolvidos, das atividades e dos produtos gerados pelas atividades.

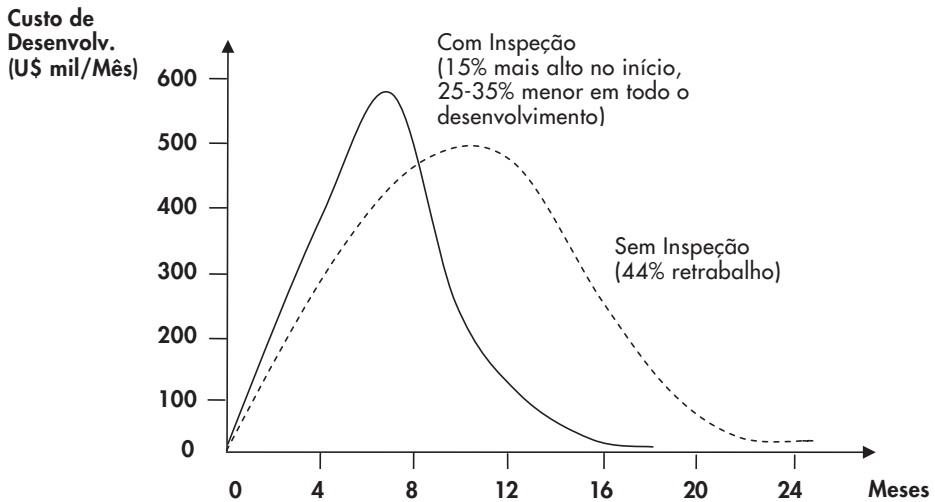


Figura 6.3 – Perfil de custos de desenvolvimento de software.<sup>42</sup>

Os papéis e a quantidade de pessoas envolvidos em um Processo de Inspeção são, segundo Sommerville:<sup>36</sup>

- a) **Autor** (uma ou mais pessoas)
  - Responsável pelo programa ou documento.
  - Fornece esclarecimentos durante a reunião de inspeção.
  - Responsável pela correção de defeitos descobertos durante o processo de inspeção.
- b) **Inspetor** (três a quatro pessoas)
  - Encontra defeitos, omissões e inconsistências em programas e documentos.
  - Pode identificar questões mais amplas fora da abordagem da inspeção.
- c) **Leitor** (uma pessoa)
  - Lê o código ou documento numa reunião de inspeção.
  - Destaca partes importantes.
- d) **Relator** (uma pessoa)
  - Documenta problemas, decisões e recomendações.
  - Registra os resultados da reunião de inspeção.

e) **Moderador** (uma pessoa)

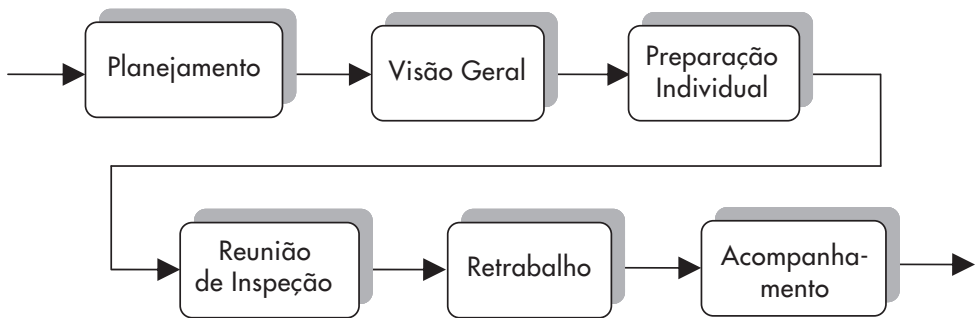
- Gerencia o processo e facilita a inspeção.
- Distribui material e agenda a reunião de inspeção.
- Acompanha o *status* das correções.
- Decide por nova reunião de inspeção.
- Relata os resultados de processo ao moderador-chefe.

f) **Moderador-chefe** (uma pessoa)

- Responsável pelas melhorias no processo de inspeção, pela atualização de *checklists* de inspeção, pelo desenvolvimento de padrões etc.

Segundo Fagan,<sup>41, 43</sup> o processo de inspeção de software é constituído das seguintes atividades, conforme mostra a Figura 6.4:

- a) **Planejamento:** O moderador é responsável por selecionar uma equipe de inspeção, organizar uma sala de reuniões e garantir que o material a ser inspecionado e suas especificações estejam completos.
- b) **Visão Geral:** O produto de software a ser inspecionado é apresentado à equipe de inspeção (inspetores), em que o autor descreve o seu conteúdo.
- c) **Preparação Individual:** Cada membro da equipe de inspeção estuda o produto e procura defeitos.
- d) **Reunião de Inspeção:**
  - A inspeção deve ser relativamente curta (com duração de não mais de duas horas).
  - Deve se ocupar, com exclusividade, em identificar defeitos, anomalias e não conformidades com padrões.
  - A equipe de inspeção não deve sugerir como esses defeitos devem ser corrigidos nem recomendar modificações em outros componentes.
- e) **Retrabalho:** O produto é modificado pelo seu autor, a fim de corrigir os problemas identificados.
- f) **Acompanhamento:**
  - O moderador deve decidir se uma nova inspeção do código é necessária. Como alternativa, ele pode decidir que uma nova inspeção completa não é necessária e que os defeitos foram corrigidos com sucesso.
  - O produto é aprovado pelo moderador para liberação.



**Figura 6.4** – Processo de inspeção, segundo Fagan.<sup>41, 43</sup>

As atividades do Processo de Inspeção geram os seguintes resultados, segundo Pressman:<sup>3</sup>

- **Registros da Inspeção:** contêm uma lista de defeitos detectados, identificando as partes problemáticas do produto; um relatório resumido de inspeção, fornecendo dados sobre a equipe de inspeção e o produto inspecionado, e as conclusões levantadas na inspeção.
- **Conclusões da Inspeção:** os participantes do processo de inspeção aceitam o produto sem modificações adicionais; os participantes rejeitam o produto devido a defeitos graves ou os participantes o aceitam provisoriamente (defeitos menores foram encontrados e devem ser corrigidos).

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

O Processo de Inspeção exige recursos e pessoas com experiência. Isso significa que, se for parcialmente implementado, o processo pode se tornar dispendioso e pouco efetivo. O papel do moderador é fundamental para se chegar a conclusões mais rápidas e objetivas. Os inspetores devem estar comprometidos para que os produtos sejam analisados com cuidado e que sejam levantadas questões realmente importantes para a melhoria da qualidade do produto.

O Processo de Inspeção pode ser realizado em projetos de qualquer tamanho. Para projetos de grande porte, seria interessante ter uma equipe de inspeção treinada e dedicada. Para projetos de pequeno porte, nem sempre esta opção é viável. Assim, recomenda-se que haja cooperação de recursos entre projetos para tentar tornar o Processo de Inspeção o mais independente possível.

Da mesma maneira que em outros processos, o Processo de Inspeção precisa ser gerenciado e melhorado. Neste caso, o moderador-chefe deve se incumbir de analisar os Processos de Inspeção realizados e propor melhorias.

## EXERCÍCIOS

1. O Processo de Inspeção procura analisar criticamente o produto alvo. Nem sempre é agradável receber uma crítica, quando se está no papel do autor. Quais recomendações poderiam ser feitas para que uma crítica não seja levada para o lado pessoal?
2. O Processo de Inspeção poderia ter sido útil em algum projeto que você tenha participado? Que fatos poderiam ser minimizados?
3. O papel do moderador poderia ser desempenhado pelo gerente de projetos? Por quê?
4. Baseando-se no Processo de Inspeção da Figura 6.4, em que fase(s) os inspetores estão envolvidos diretamente? Justifique.
5. Em que fase(s) do processo de desenvolvimento deve(m) ser realizada(s) as inspeções? Justifique. (Use o processo Cascata como referência.)
6. Por que as inspeções podem aumentar a produtividade de desenvolvimento de software?
7. De que maneira as inspeções podem melhorar o gerenciamento de projetos de software?
8. Suponha que você irá participar de um processo de inspeção, cujo produto é uma especificação de requisitos baseada na norma IEEE 830 [74]. Elabore duas questões que devem ser analisadas nesta especificação.
9. Sugira como um Processo de Inspeção pode ser aplicado em processos ágeis como o *Extreme Programming* (XP).



10. Um bom caso de teste é aquele que tem uma elevada probabilidade de revelar um defeito ainda não descoberto.<sup>33</sup> De que maneira uma inspeção pode descobrir defeitos em programas e documentos em geral?

### SUGESTÕES DE LEITURA

ACKERMAN, A. Frank; BUCHWALD, Lynne S.; LEWSKI, Frank H. Software inspections: an effective verification process. *IEEE Software*, vol. 6, issue 3, pp. 31-36, 1989.

FAGAN, Michael E. Design and code inspections to reduce errors in program development. *IBM System Journal*, vol. 15, no. 3, pp. 182-211, 1976  
FAGAN, Michael E. Advances in software inspection. *IEEE Transactions on Software Engineering*, vol. 12, issue 7, pp. 744-751, Jul., 1986.

PARNAS, David L.; LAWFORD, Mark. The role of inspection in software quality assurance. *IEEE Transactions on Software Engineering*, vol. 29, issue 8, pp. 674-676, August, 2003.

PORTER, Adam; VOTTA, Lawrence. What makes inspections work? *IEEE Software*, vol. 14, issue 6, pp. 99-102, Nov., 1997.

WHEELER, David A.; BRYKCZYNSKI, Bill.; MEESON Jr., Reginald. N. *Software inspection – an industry best practice*. USA: IEEE Computer Society Press, 1996.

# Configuração

---

Uma Configuração é o conjunto de itens constituintes de um sistema. Dada a complexidade dos sistemas e a quantidade destes itens, é importante saber em um determinado instante qual é a sua configuração. Esta informação é usada no desenvolvimento e na operação do software. No desenvolvimento, ela permite saber quais são os impactos ao projeto quando uma solicitação de mudança é recebida. Também no momento de manutenção, ela permite que se gerem novos *releases* (uma versão particular de um item de configuração para um propósito específico) que incorpore uma nova funcionalidade. Para isso, a disciplina de Gerenciamento de Configuração, com ênfase em Gerenciamento de Mudanças, é importante para manter o desenvolvimento sob controle.

---

## 7.1. GERENCIAMENTO DE CONFIGURAÇÃO

*A disciplina de Engenharia de Software possibilitou o surgimento de abordagens de desenvolvimento e manutenção de software. Desde o início, os processos, métodos e as técnicas foram criados para desenvolver softwares cada vez mais complexos. Durante o desenvolvimento, muitos produtos intermediários são gerados e as equipes de desenvolvimento usam estes produtos para chegar ao produto final. Assim, esses produtos intermediários devem ser controlados de maneira que versões desatualizadas não sejam usadas. A disciplina de Gerenciamento de Configuração é fortemente relacionada à garantia de qualidade de software. Enquanto a garantia de qualidade*

*fornece garantia de que os produtos e processos de software atendem aos requisitos especificados, o gerenciamento de configuração procura garantir a integridade dos produtos durante o desenvolvimento e, também, na manutenção.*

Um sistema pode ser definido como uma coleção de componentes organizados para realizar uma função específica ou um conjunto de funções.<sup>2</sup> Uma configuração é um conjunto de versões desses componentes. De forma mais geral, também pode ser uma coleção de versões específicas de itens de hardware, firmware ou software combinados de acordo com procedimentos de construção para servir a um propósito particular.<sup>7</sup>

Os processos de desenvolvimento de software geram produtos durante a execução de suas fases. Cada produto é gerado por um responsável alocado pelo gerente de projetos. Por exemplo, a Especificação de Requisitos de Software é gerada por um analista de sistemas, a Especificação de Projeto por um arquiteto, os Códigos (fonte e executável) pelos programadores e, assim por diante. Cada produto deve ser mantido em um repositório do projeto de maneira que possa ser usado pelos envolvidos no projeto. A questão que surge é como saber se esse produto está íntegro e na sua última versão. Seria muito danoso e oneroso para o projeto se algum envolvido usasse um produto que não atende a esses critérios.

Assim, há a necessidade de ter um controle sobre estes produtos e suas versões. A disciplina de Gerenciamento de Configuração foi criada com esse propósito. Dentro desta disciplina existem alguns conceitos importantes que devem ser entendidos para melhor execução de suas atividades.

Na disciplina de Gerenciamento de Configuração os produtos gerados são conhecidos como Itens de Configuração de software (SCI – Software Configuration Item) que é um conceito fundamental dessa disciplina. Os SCI têm uma relação direta com os produtos gerados pelos envolvidos no projeto. Entende-se como Item de Configuração um agrupamento de produtos coesos que fazem sentido para o projeto. Os SCI típicos são Especificações (de Requisitos, de Projeto, de Código etc.) e suas respectivas

representações (diagramas, gráficos etc.), Planos (de Projeto, de Testes, de Configuração, de Garantia de Qualidade etc.), Manuais (de Usuário, de Operação, Implantação, Operação etc.), Programas (Códigos-fonte, Executáveis), Relatórios (de Teste, de Operação etc.), Padrões e Procedimentos, Ferramentas (Compiladores, de Teste, Depuradores etc.).

O segundo conceito fundamental é o de *Baseline*. Para ilustrar esse conceito, pode-se pensar na figura de uma porta para a cozinha de um famoso restaurante, ou seja, de um lado a cozinha e do outro o local onde ficam os clientes.<sup>3</sup> Quando um requisito é especificado pelo cliente (entenda-se o “prato”), o garçom envia a especificação para a cozinha. O chefe da cozinha encarrega-se de criar o prato conforme especificado. Durante a criação, podem surgir diversas versões do prato, até que no fim surge aquele que atende ao requisito do cliente. Uma vez que o prato é verificado e aprovado, estabelece-se o conceito de *Baseline*. Para o prato, na sua última versão verificada e aprovada, é estabelecida uma marca de tempo que registra essa versão como a final nas condições citadas. Ao entregar para o cliente, se o requisito não for atendido, o prato deverá voltar para a cozinha, senão ficará com o cliente. A eventual correção do prato deve ser formalmente solicitada através de uma Solicitação de Mudança que deverá ser analisada pelo chefe da cozinha.

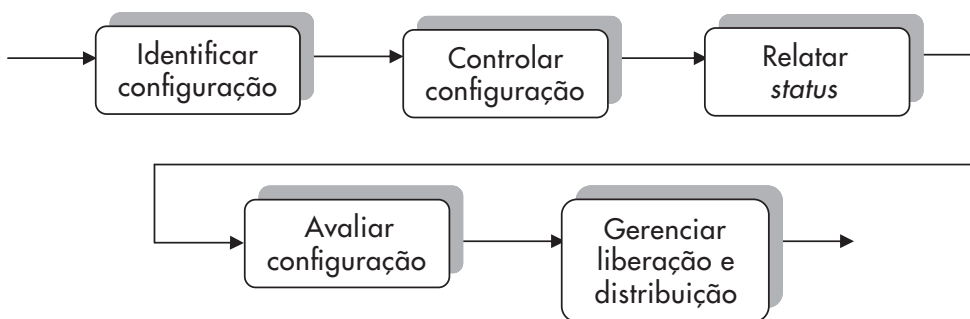
O terceiro conceito é então a Solicitação de Mudança. Toda mudança de algum SCI, que estiver em algum *Baseline*, só poderá ser alterada, seguindo-se um procedimento padrão definido. O responsável pela análise e aprovação da Solicitação de Mudança é conhecido como Comitê de Controle de Configuração (CCB – *Configuration Control Board*). (veja mais detalhes no Capítulo 7).

Segundo Pressman,<sup>3</sup> Gerenciamento de Configuração de software é um conjunto de atividades desenvolvido para administrar as mudanças dentro do ciclo de vida do software. Já Sommerville<sup>8</sup> define o Gerenciamento de Configuração de software como o desenvolvimento e a aplicação de procedimentos para gerenciar um produto de sistema em desenvolvimento. No IEEE 12207,<sup>44, 45, 46</sup> o Gerenciamento de Configuração é uma

disciplina voltada a identificar, definir e gerar *baseline* de itens de software em um sistema, controlar mudanças e *releases* (versão particular de item de configuração gerado para um propósito específico) de itens, registrar e reportar *status* de itens e solicitações de mudanças, assegurar completeza, consistência e correção dos itens e controlar armazenamento, manipulação e entrega de itens em todo o ciclo de vida do software.

As atividades do Gerenciamento de Configuração,<sup>44, 45, 46</sup> conforme a Figura 7.1, são as seguintes:

- Identificar configuração: Deve-se definir uma sistemática para o projeto de identificação dos itens de software e as versões a serem controladas.
- Controlar configuração: Deve-se identificar e registrar as solicitações de mudança, análises e avaliações das mudanças, aprovação ou rejeição do pedido e implementação, verificação e liberação de um item de software modificado. Além disso, devem existir registros de auditoria para que as mudanças possam ser rastreadas.
- Relatar *status* de configuração: Devem-se preparar registros de gerenciamento e relatório de *status* que mostrem a situação e o histórico dos itens de software controlado, incluindo os *baselines*.
- Avaliar configuração: Deve-se garantir a completeza funcional dos itens de software em relação aos seus requisitos e a completeza física dos itens de software.
- Gerenciar liberação e distribuição: Deve-se controlar formalmente a liberação e a distribuição de produtos de software e a sua documentação.



**Figura 7.1** – Atividades de Gerenciamento de Configuração.

Os papéis normalmente desempenhados na disciplina de Gerenciamento de Configuração são:

- Gerente de Configuração: pode ser um analista de configuração ou o próprio gerente de projetos.
- CCB: pode ser formado pelo representante dos desenvolvedores, gerente de projeto e em alguns casos, alguém da alta gerência e o próprio cliente.

Para que a disciplina de Gerenciamento de Configuração funcione, é necessário ter um plano para orientar suas atividades. Segundo Humphrey,<sup>38</sup> uma estrutura de Plano de Gerenciamento de Configuração possui as seguintes seções:

- a) Visão Geral  
Nesta seção definem-se os objetivos e o sistema de software que estará sob controle de configuração.
- b) Organização  
Nesta seção definem-se as responsabilidades relativas ao CCB, Garantia de Qualidade, Gerenciamento de Configuração, Auditoria de Configuração etc.
- c) Métodos e Técnicas  
Nesta seção definem-se os métodos e as técnicas para atualização de versões, geração e liberação de *baselines*, geração de *releases*, geração de relatórios de *status* etc.
- d) Procedimentos  
Nesta seção definem-se os procedimentos de solicitação e acompanhamento de mudanças e o formulário a ser usado.
- e) Implementação  
Nesta seção definem-se os marcos de projeto, infraestrutura necessária etc.

Para realizar as diversas atividades previstas na disciplina de Gerenciamento de Configuração, ela deve ser estruturada por meio de um processo definido pela organização de software.

Existem padrões e modelos reconhecidos que podem auxiliar na estruturação de um processo, seguindo-se as boas práticas descritas nesta seção.

A ISO e a ANSI (*American National Standards Institute*)/IEEE definiram os seguintes padrões para esta disciplina:

- ISO ANSI/IEEE 828-1983;<sup>47</sup>
- ISO ANSI/IEEE 1042-1987.<sup>48</sup>

Alguns modelos que têm se tornado um padrão na indústria de software, como o CMMI<sup>49, 50</sup> e MR-MPS,<sup>17, 51, 52</sup> possuem processos relacionados ao Gerenciamento de Configuração.

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

O gerente de projeto ao elaborar um plano de desenvolvimento de software deve definir uma estratégia de geração de versões de produtos e *releases*. No primeiro caso, os produtos são produtos intermediários que serão gerados durante a execução das fases do projeto. Cada produto intermediário pode ter inúmeras versões. Entende-se como versão uma instância do sistema que difere de muitas maneiras de outras instâncias. No segundo caso, o gerente pode ter definido que o sistema de software será entregue de forma incremental. Cada incremento representa um *release* que será distribuído ao cliente.

A disciplina de Gerenciamento de Configuração apoia o gerenciamento de projetos e possibilita que as equipes de desenvolvimento estejam usando a versão correta do software, rastreiem as mudanças de software ocorridas ao longo do desenvolvimento e que a organização gerencie melhor os ativos de software. Como consequência, melhora-se a qualidade do software.

Essa disciplina é fortemente relacionada à Garantia de Qualidade de software. Enquanto a Garantia de Qualidade fornece garantia de que os produtos e processos de software atendem aos requisitos especificados (veja mais detalhes no Capítulo 8), o Gerenciamento de Configuração procura garantir a integridade dos produtos durante o desenvolvimento e, também, na manutenção.

Ela requer diversas atividades para manter a integridade dos produtos gerados durante o processo de desenvolvimento de software. Embora existam diversas atividades, pode-se notar também que existem boas práticas que permitem atingir os objetivos de uma boa disciplina de Gerenciamento de Configuração.

Entretanto, a existência de padrões e modelos aceitos pela indústria de software permite uma boa estruturação de um processo para a organização de software.

No mundo atual, o crescimento da complexidade das soluções exige cada vez mais uma disciplina de Gerenciamento de Configuração de software.

## EXERCÍCIOS

1. É possível desenvolver um software sem uma disciplina de Gerenciamento de Configuração de software? Justifique.
2. Quais são as dificuldades de desenvolver um software a partir de componentes?
3. Identifique os itens de configuração de um processo de software com o qual você esteja familiarizado.
4. O que você entende por *baseline*? Qual é a sua importância?
5. Defina os *baselines* a partir de itens de configuração do processo usado no exercício 3.
6. Pesquise uma ferramenta de gerenciamento de configuração e descreva as suas funções mais importantes.
7. Usando a ferramenta de gerenciamento pesquisada, descreva como as versões de produtos são criadas.
8. Usando a mesma ferramenta, descreva como as *baselines* são criadas.
9. Explique com exemplos como a disciplina de Gerenciamento de Configuração está relacionada à Garantia de Qualidade de software?
10. Suponha que você irá definir uma disciplina de Gerenciamento de Configuração na sua empresa. Quais são as premissas que você deve assumir?



## SUGESTÕES DE LEITURA

BERSOFF, Edward H.; HENDERSON, Vilas D.; SIEGEL, Stanley G. Software configuration management: a tutorial. *IEEE Computer*, vol. 12, issue 1, pp. 6-14, Jan., 1979.

BERSOFF, Edwards H.; DAVIS, Alan M. Impacts of life cycle models on software configuration management. *Communications of the ACM*, vol. 34, issue 8, pp. 105-118, Aug., 1991.

BOURQUE, Pierre.; DUPUIS, Robert.; TRIPP, Leonard. L. (Eds.). *Guide to the software engineering body of knowledge – SWEBOK*. USA: IEEE Computer Society Press, 2004.

CHRISSIS, Mary B.; KONRAD, Mike; SHRUM, Sandra. *CMMI for development – guidelines for process integration and product improvement*. 3<sup>rd</sup> edition. USA: Addison-Wesley Professional, 2011.

FELDMAN, Stuart I. Software configuration management – past uses and future challenges. *Lecture Notes in Computer Science*. vol. 550, pp. 1-6, 1991.

HUMPHREY, Watts S. *Managing the software process*. USA: Addison-Wesley Professional, 1989.

IEEE. *ANSI/IEEE Std 828-1990 – IEEE standard for software configuration management plans*. New York: IEEE, 1990.

IEEE. *ANSI/IEEE Std 1042-1987 – IEEE guide to software configuration management*. New York: IEEE, 1987.

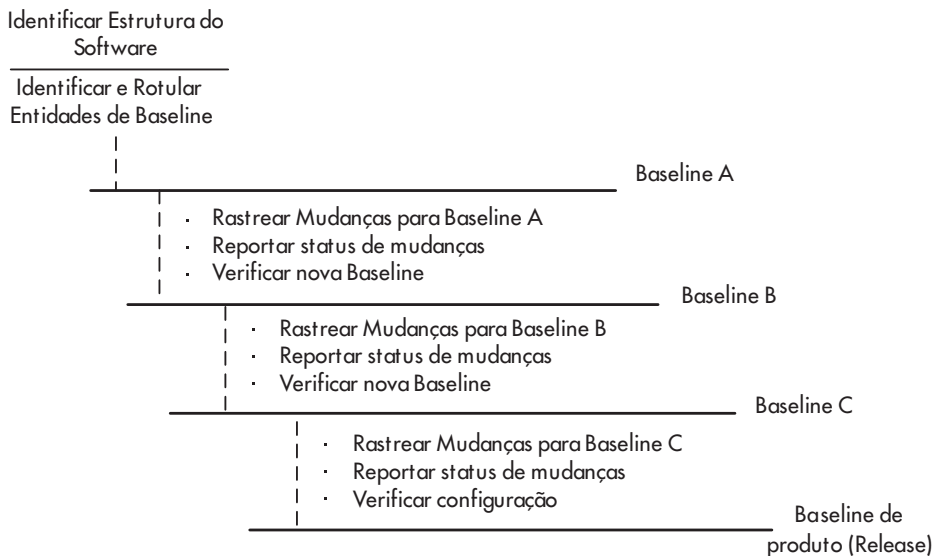
## 7.2. GERENCIAMENTO DE MUDANÇAS

*Em todo o desenvolvimento de software ocorrem mudanças. Existem vários motivos pelos quais o software precisa ser mudado. Uma das razões é a mudança de requisitos solicitada pelo cliente ou a mudança em soluções solicitada pelos desenvolvedores. Na primeira, o cliente solicita uma mudança por inclusão de novos requisitos, ou alteração ou retirada de requisitos já acordados. Normalmente, essas mudanças ocorrem por alterações nos negócios. Na segunda, decisões inadequadas de projeto ou defeitos detectados levam à necessidade de mudanças. Todas essas mudanças precisam ser gerenciadas pela simples razão da necessidade de manter a rastreabilidade das mudanças de versões de produtos e não chegar ao descontrole do desenvolvimento de software.*

Como já foi citado anteriormente, para atender ao objetivo de manter a integridade dos produtos armazenados em repositórios do projeto, o Gerenciamento de Configuração convive com a necessidade de mudanças. As mudanças ocorrem de várias maneiras. A mais comum é a mudança de requisitos de software. Grande parte dessas mudanças ocorre nas áreas de negócio do cliente que, por várias razões (estratégicas, legais, mercados etc.), necessita alterar um ou alguns requisitos definidos para o software em desenvolvimento. A outra maneira é quando algum membro do projeto precisa mudar alguma definição anterior (por exemplo, interfaces da arquitetura do software). Em ambos os casos, não se pode prever quando essas mudanças serão solicitadas. No entanto, dependendo das análises de impacto no projeto, e se for aprovado pelo CCB, a mudança deverá ocorrer. Conforme foi descrito anteriormente, essa mudança ocorrerá seguindo-se um procedimento formal através de uma Solicitação de Mudança.

O Gerenciamento de Mudanças (ou controle de configuração) está relacionado a todas as mudanças de itens de configuração que estão em algum baseline, durante todo o ciclo de vida do software. A Figura 7.2 apresenta um modelo de Gerenciamento de Mudanças baseado em baselines ao longo do desenvolvimento. A rastreabilidade das mudanças obtida durante o desenvolvimento permite que o software seja mantido durante a sua operação.

O Gerenciamento de Mudanças inicia com a identificação da estrutura do software. Essa estrutura inicial evolui ao longo do desenvolvimento e cada baseline incorpora as mudanças solicitadas até a liberação do software. O relatório de status permite verificar a situação e o histórico dos itens de configuração e os respectivos baselines.



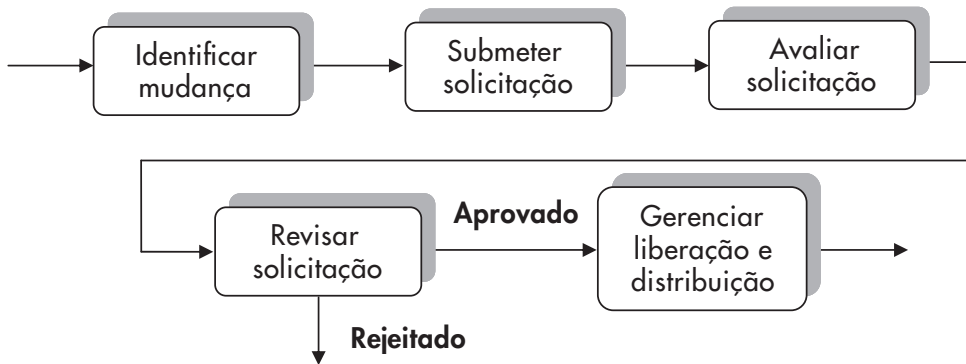
**Figura 7.2** – Modelo de Gerenciamento de Mudanças.<sup>48</sup>

O Gerenciamento de Mudanças contém as seguintes atividades (adaptado)<sup>7</sup>, conforme mostra a Figura 7.3:

- a) Identificar mudança: requerente identifica a mudança.
- b) Submeter Solicitação de Mudança: requerente submete mudança.
- c) Avaliar solicitação: gerente de configuração avalia a completeza da solicitação.
- d) Revisar solicitação: CCB revisa solicitação e decide:
  - se rejeitado, informa requerente;
  - se aprovado, designa engenheiro de software;

- e) Programar, projetar, testar: engenheiro de software implementa a mudança.
- f) Fechar mudança: gerente de configuração fecha a solicitação.

Para haver um bom sincronismo entre as atualizações dos produtos, pode-se notar que um mecanismo presente em várias ferramentas de apoio possui os conceitos de *Check-in*, *Check-out* e Controle de Acesso baseados em um repositório de SCI (Item de Configuração de Software).



**Figura 7.3** – Atividades de Gerenciamento de Mudanças.

Quando uma mudança deve ser efetuada em algum SCI, deve-se ativar um *Check-out* do SCI desejado no repositório do projeto. Ao retirar o SCI, o *Check-out* deve sinalizar ao Controle de Acesso que um determinado SCI foi retirado e deve-se bloquear o acesso para algum outro envolvido no projeto. Este SCI então é passado para a área de Engenharia que fará as mudanças conforme aprovadas na Solicitação de Mudança. Ao efetuar as mudanças, um *Check-in* é ativado para que o novo SCI retorne ao repositório. Ao mesmo tempo, o *Check-in* sinaliza ao Controle de Acesso para desbloquear o acesso ao novo SCI.

Para garantir que as mudanças foram adequadamente implementadas podem-se usar algumas técnicas de verificação usadas no processo de desenvolvimento do software. Técnicas de Inspeção podem ser muito bem aplicadas nesse caso. Outra técnica é aquela prevista no próprio

Gerenciamento de Configuração, que é a Auditoria de Configuração de software.

Uma Solicitação de Mudança é realizada normalmente por meio de um formulário padrão definido. Esse formulário pode apresentar as seguintes informações:

- Identificação do sistema de software.
- SCI (Item de Configuração de Software), objeto da mudança.
- Autor original do SCI.
- Data da solicitação.
- Sistema, subsistemas e/ou módulos afetados.
- Descrição da mudança.
- Análise de impactos ao sistema/projeto de software.
- Prazo previsto para implementação da mudança.
- Aprovação da mudança pelo CCB (conforme definido no Plano de Gerenciamento de Configuração).

Durante a execução do Plano de Gerenciamento de Configuração, uma Auditoria de Configuração deve ser realizada. A auditoria tem como objetivo verificar se todas as atividades previstas no plano foram seguidas, em última análise, se os SCI estão íntegros no repositório de configuração do projeto.

Uma auditoria é realizada por alguém designado pelo gerente de projeto, ou em alguns casos, pela própria organização de software, dependendo da criticidade do software para os negócios ou dos riscos para a organização e/ou para o cliente e usuários.

Uma auditoria segue um procedimento definido, que é de conhecimento, a priori, por todos os envolvidos. Esse procedimento pode ser definido por meio de um *checklist*, conforme segue:

- A mudança especificada foi efetuada?
- Modificações adicionais foram incorporadas?

- As solicitações de mudança foram fechadas?
- As mudanças efetuadas foram aprovadas?
- Revisões ou inspeções de software foram realizadas?
- Procedimentos de Gerenciamento de Configuração foram seguidos?
- Padrões de desenvolvimento foram seguidos?
- Existem registros de todas as atividades de gerenciamento de versões?

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

Durante todo o ciclo de vida do software, desenvolvimento e operação, vão ocorrer mudanças. Existem vários motivos pelos quais o software precisa ser mudado. Uma das razões é a mudança em requisitos solicitada pelo cliente ou a mudança em soluções solicitada pelos desenvolvedores ao longo do desenvolvimento ou correções e melhorias no software solicitadas pelo cliente durante a sua operação.

O cliente solicita uma mudança por inclusão de novos requisitos, ou alteração ou retirada de requisitos já acordados. Normalmente, isso ocorre por alterações nos negócios. As decisões inadequadas tomadas durante o desenvolvimento ou defeitos detectados em testes também levam à necessidade de mudanças. Além disso, durante a operação, melhorias são necessárias. Todas essas mudanças precisam ser gerenciadas pela simples razão da necessidade de manter a rastreabilidade das mudanças de versões de produtos e de não cair no descontrole do desenvolvimento de software.

Toda mudança é realizada por meio de um procedimento formal. A submissão de mudança é feita por um formulário denominado Formulário de Solicitação de Mudança (CRF – *Change Request Form*). A aprovação ou não da solicitação é feita pelo CCB. O CCB é um grupo responsável pelas decisões de mudanças nos *baselines*. A composição do CCB é muito

variada dentro das empresas, que vai desde o gerente de projeto à área de alta gerência e de desenvolvimento da organização e, em alguns casos, até o cliente.

O Gerenciamento de Mudanças é feito normalmente com apoio de ferramentas de Gerenciamento de Configuração, tais como o CVS (*Concurrent Versions System*) que é um software *open-source* nos termos da sociedade Open Source Initiative, ClearCase da empresa Rational/IBM, StarTeam da empresa Borland e SourceSafe da empresa Microsoft.

## EXERCÍCIOS

1. O Gerenciamento de Mudanças está associado ao Gerenciamento de Configuração. O que você entende por mudança em desenvolvimento de software?
2. Por que uma mudança deve ser solicitada formalmente por meio de um formulário padrão CRF (*Change Request Form*)?
3. Escolha um projeto que você tenha participado e descreva como as mudanças foram tratadas. Havia alguém no papel do CCB?
4. No projeto do item anterior, quais foram os *baselines* definidos? Eles foram criados?
5. Na sua empresa, como as novas versões e *baselines* seriam comunicadas para as equipes de desenvolvimento?
6. Toda mudança altera um item de configuração que está em algum *baseline* definido no projeto. Como o item de configuração é incorporado ao projeto após a mudança?
7. Descreva como o atendimento de uma Solicitação de Mudança pode ser verificado.
8. Em sua opinião, quando uma Solicitação de Mudança deve ser rejeitada?
9. Descreva a importância da Auditoria de Configuração.
10. O Gerenciamento de Mudanças discutido nesta seção e na seção 3.2 são semelhantes. Descreva quais são as similaridades e as diferenças.

## SUGESTÕES DE LEITURA

BOURQUE, Pierre; DUPUIS, Robert; TRIPP, Leonard L. (Eds.). *Guide to the software engineering body of knowledge – SWEBOK*. USA: IEEE Computer Society Press, 2004.

CHRISSIS, Mary B.; KONRAD, Mike; SHRUM, Sandra. *CMMI for development – guidelines for process integration and product improvement*. 3<sup>rd</sup> edition. USA: Addison-Wesley Professional, 2011.

HUMPHREY, Watts S. *Managing the software process*. USA: Addison-Wesley Professional, 1989.

PRESSMAN, Roger S. *Software engineering – a practitioner’s approach*. 6<sup>th</sup> edition. New York: McGraw Hill, 2007.

SOMMERVILLE, Ian. *Engenharia de Software*. Tradução: Kalinka Oliveira e Ivan Bosnic. Revisão Técnica: Kechi Hiramã. 9. ed. São Paulo: Prentice Hall, 2011.



# Qualidade

---

Qualidade é algo difícil de ser definido. Ainda mais difícil é garantir a qualidade de algum produto. Cada pessoa tem uma percepção diferente sobre qualidade. Depende das expectativas de cada um sobre algum produto ou serviço. Se as expectativas são atendidas de alguma maneira, é possível dizer que o produto ou serviço tem qualidade. Do ponto de vista de objetivo, garantia de qualidade se assemelha ao V&V, visto anteriormente. Ambos devem ter o caráter de independência. A diferença é que V&V está associado ao produto ou serviço, de forma que atenda às especificações, e garantia de qualidade está associada aos produtos: se foram produzidos conforme padrões definidos, quais foram os processos realizados e se seguiram os processos definidos.

---

### 8.1. INTRODUÇÃO À QUALIDADE

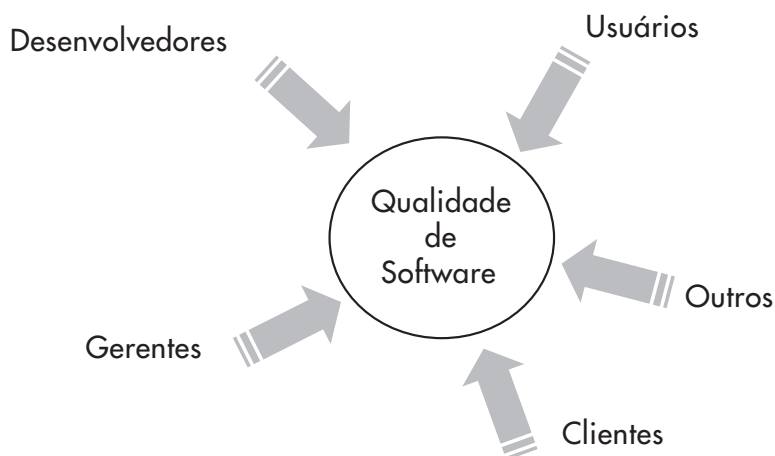
*A questão sobre a necessidade de qualidade em software vem sendo colocada há mais ou menos 40 anos. Se qualidade depende de processos de desenvolvimento definidos, pessoas capacitadas e treinadas e ferramentas de apoio, um cenário adequado nessas condições já está disponível há um bom tempo. Uma segunda questão seria "Por que ainda existem tantos defeitos no software?". Comparativamente, a qualidade está bem resolvida, por exemplo, na indústria de manufatura. A área possui normas e padrões de qualidade que regulam o nível de qualidade requerido para um produto*

*ser aceito pelos usuários. No caso do desenvolvimento de software, existem normas e padrões específicos, porém o software ainda é produzido com muitos defeitos.*

Qualidade é algo que todos querem, porém é muito difícil defini-la. Segundo Hetzel,<sup>32</sup> a qualidade é não tangível. Já para Juran<sup>4</sup> qualidade é a satisfação das necessidades do consumidor... e a adequação ao uso. Crosby<sup>4</sup> acredita que qualidade é a conformidade às especificações que prevenir não conformidades é mais barato que corrigir ou refazer o trabalho. Chrissis, Konrad e Shrum<sup>49</sup> disseram que a qualidade é a capacidade de um conjunto de características inerentes de um produto, componente ou processo para atender aos requisitos dos clientes.

Pelas definições apresentadas, a qualidade é algo intangível presente em um produto. Em outra perspectiva, pode-se dizer que a qualidade é algo que deve ser conquistado e não é simplesmente obtido. Há necessariamente um esforço a ser dispendido. O software é um produto resultante de uma necessidade do cliente, não possui forma física, mas ocupa um espaço na memória do computador e embute uma série de tecnologias. O software, como produto, também precisa ter qualidade. Então, afinal, o que é qualidade de software? A Figura 8.1 sintetiza uma forma de responder a essa questão.

A resposta é baseada nas expectativas dos *stakeholders*. Do ponto de vista dos desenvolvedores, a qualidade pode ser vista como atendimento a métodos e padrões de software. No caso dos gerentes, que o projeto fique próximo aos parâmetros estimados de projeto (esforço, custo e prazo). No caso de usuários, que o software seja de fácil uso e compreensão. No caso de clientes, que o software atenda às suas necessidades de negócio e aos prazos e custos acordados. No caso de outros, por exemplo, gerenciamento de configuração, que as versões do software estejam íntegras ao longo do desenvolvimento.



**Figura 8.1** – *Stakeholders* e suas expectativas sobre qualidade de software.

O termo “qualidade” tem raízes na indústria de manufatura desde a Revolução Industrial no século XVIII. No início, deu-se prioridade à produção em massa e somente em 1924, com a criação dos conceitos de controle estatístico de processo (CEP) e de ciclo PDCA (*Plan-Do-Check-Act*) por Walter A. Shewhart, a qualidade evoluiu bastante. A melhoria de qualidade também passa pelo período pós-guerra entre 1950 e 1960 no Japão com W. Edwards Deming e Joseph M. Juran que introduziram os conceitos de CEP e de Gerenciamento da Qualidade.

O Gerenciamento da Qualidade Total (TQM – *Total Quality Management*) tem início em 1980 com ênfase no gerenciamento organizacional.<sup>4, 5</sup>

O TQM é uma abordagem gerencial (princípios, métodos e técnicas) para obter sucesso em longo prazo através da satisfação dos clientes. Na realidade, a satisfação dos clientes está ligada diretamente ao planejamento estratégico da empresa. Melhoria contínua da qualidade e aumento de produtividade fazem parte das metas deste planejamento.

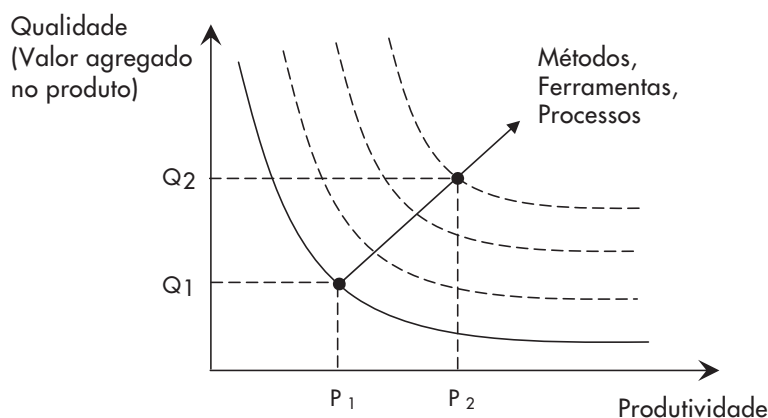
O Gerenciamento da Qualidade de software tem sua origem no TQM. Portanto, a busca pela qualidade de software não é recente. No entanto, pode-se dividir o desenvolvimento de software em dois momentos: a fase

do software artesanal e a do software profissional. Na fase artesanal, antes da definição da Engenharia de Software em 1969, o software era em geral desenvolvido e testado continuamente até se chegar a algo que funcionasse e que o cliente pudesse aceitar. Na fase profissional, já na era da Engenharia de Software, além de funcionar e ser aceito pelo cliente, o software precisava ser padronizado, documentado e com boa relação custo/benefício.

A qualidade sempre foi desejada e inquestionável. Quem aceitaria um software de má qualidade ou que fosse duvidoso quanto ao seu funcionamento? No entanto, a obtenção da qualidade também passa por alguns obstáculos que não residem somente no aspecto técnico em obtê-lo, embora a complexidade das soluções aumente a cada ano. O principal obstáculo está relacionado à cultura da organização. A qualidade é fortemente influenciada pela cultura ou resistência a mudanças da organização.

Assim, os fatores importantes para obter qualidade são ter processos adequados e ter pessoas bem treinadas e capacitadas para exercerem a contento as suas atividades. Atualmente, não atender a esses fatores é indicador de má qualidade do software.

Porém, deve haver também uma boa produtividade. Somente processos e pessoas adequadas não alcançam por si só a produtividade desejada. A Figura 8.2 ilustra a relação entre qualidade *versus* produtividade do ponto de vista de valor agregado no produto. Ter alta qualidade significa ter alto valor agregado nos produtos sem considerar os níveis de produtividade e, por outro lado, ter alta produtividade significa ter muitos produtos sem considerar os níveis de qualidade. Assim, um produto com qualidade é mais caro do que aquele obtido somente com produtividade. Atualmente, bons níveis de qualidade e produtividade são essenciais em empresas competitivas. Portanto, pode-se pensar em um ponto médio da curva como ideal. Porém, neste ponto, a qualidade e a produtividade são baixas. Nesse caso, deve-se afastar a curva dos eixos e melhorar a posição do ponto médio para se obter qualidade e produtividade em níveis mais altos. Para tanto, é necessário aplicar métodos reconhecidos, ferramentas testadas e processos que incorporem boas práticas consagradas.



**Figura 8.2** – Relação entre qualidade e produtividade.

Atualmente é muito comum as empresas obterem certificados de qualidade. Neste cenário, existem modelos e padrões de qualidade de software que podem ser usados para melhorar os processos das empresas. Os mais conhecidos são o padrão ISO 9001<sup>53, 54</sup> e os modelos CMMI e MR-MPS (veja mais detalhes no Capítulo 11).

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

A qualidade requer um esforço para depurar e agregar valor ao produto. Evidentemente, isso implica em custo. A equação a ser resolvida é quanto esforço é necessário e suficiente para considerar um produto acabado e pronto para satisfazer as necessidades do cliente. Não é uma equação trivial.

Na tentativa de minimizar os esforços, mas com qualidade, tende-se a investir em ferramentas que prometem automatizar e controlar os projetos de software. Como foi visto no Capítulo 1, a camada de foco em qualidade é precedida de camadas de processos e métodos, antes de ferramentas.

Outro aspecto importante é que a qualidade deve ser conquistada (não somente obtida) por meio da melhoria contínua dos processos. O maior obstáculo para a melhoria contínua é a cultura organizacional que precisa estar aberta às novas abordagens e pensar no coletivo. Nada ajudaria se processos, métodos e ferramentas estivessem disponíveis, mas não efetivos dentro da organização.

Assim, surgiu no início da década de 1990 o conceito de maturidade de processos, com a proposta do modelo CMM (atualmente CMMI) do SEI (*Software Engineering Institute*) da Carnegie Mellon University dos Estados Unidos, baseado fortemente no TQM. Outro exemplo de modelo de maturidade é o MR-MPS administrado pela Softex, entidade do Ministério da Ciência e Tecnologia do Brasil. (veja mais detalhes no Capítulo 11).

## EXERCÍCIOS

1. Por que o software não pode ser tratado como um produto de manufatura?
2. Em um processo dito de qualidade produzem-se necessariamente produtos de qualidade? Justifique.
3. O desenvolvimento de um software de qualidade é necessariamente mais caro do que aquele não orientado à qualidade?
4. Sugira um modo de minimizar os efeitos da cultura organizacional para a melhoria de processos de software.
5. Por que se deve definir processos e métodos antes de ferramentas?
6. Em sua opinião, quanto de esforço é necessário e suficiente para atender às necessidades dos clientes?
7. Sugira como se pode atender às necessidades dos clientes, se estas mudam ao longo do desenvolvimento de software?
8. Alguns desenvolvedores de software de uma organização não seguem os processos sistematicamente, comprometendo a qualidade do software. Sendo você o gerente responsável, o que você faria? Justifique.

9. Seu chefe mandou-lhe uma caixa de listagens de código-fonte de um programa crítico da organização para que você avaliasse sua qualidade. Considerando os conceitos apresentados, como você faria esta avaliação?
10. Uma empresa de desenvolvimento de software constatou que um elevado número de defeitos é entregue aos seus clientes. Você foi contratado para resolver esse problema. O que você faria? Justifique.

## SUGESTÕES DE LEITURA

ARTHUR, Lowell. J. *Melhorando a qualidade do software – Um guia completo para o TQM*. Rio de Janeiro: IBPI Press, 1994.

BARTEL, Timothy; FINSTER, Mark. A TQM process for systems integration: getting the most from COTS software. *Information System Management*, vol. 12, issue 3, pp. 19-29, Summer, 1995.

BOEHM, Barry W. Improving software productivity. *IEEE Computer*, vol. 20, issue 9, pp. 43-57, Sep., 1987.

BOEHM, Barry W. et al. A software development environment for improving productivity. *IEEE Computer*, vol. 17, issue 6, pp. 30-42, 1984.

BOURQUE, Pierre; DUPUIS, Robert; TRIPP, Leonard L. (Eds.). *Guide to the software engineering body of knowledge – SWEBOK*. USA: IEEE Computer Society Press, 2004.

HUMPHREY, Watts S. *Managing the software process*. USA: Addison-Wesley Professional, 1989.

KENETT, Ron S.; KOENIG, Shaye. A process management approach to SQA. *Quality Progress*, vol. 21, issue 11, pp. 66-70, Nov., 1988.

CARVALHO, Marli M.; PALADINI, Edson. P. et al. *Gestão da qualidade – Teoria e casos*. Rio de Janeiro: Campus, 2006.

## 8.2. GERENCIAMENTO DE QUALIDADE

*Dadas as inúmeras variáveis existentes para obter a qualidade desejada do software, cabe falar em gerenciamento desta qualidade. Como gerenciamento, deve-se planejar o que se deseja de qualidade e, conseqüentemente, deve-se controlar os resultados de maneira a verificar se o que foi planejado está sendo obtido. Os desvios entre o planejado e o obtido denominam-se não conformidades. Elas devem ser discutidas e resolvidas dentro do projeto de software. As não conformidades devem ser encaradas como dados de reflexão para a melhoria do processo de desenvolvimento de software. Em alguns aspectos, até se o que foi planejado é realmente necessário ou factível em um determinado momento. A melhoria de processos deve ser conquistada gradativamente.*

Gerenciamento de Qualidade refere-se às atividades de apoio aos projetos para monitorar e gerenciar a qualidade do software. Incluem-se as atividades de planejamento de qualidade, definições de processos, monitoração de processos etc.<sup>55</sup>

O Gerenciamento de Qualidade pode ser organizado em basicamente três atividades:<sup>8</sup>

- Garantia de Qualidade.
- Planejamento de Qualidade.
- Controle de Qualidade.

**Garantia de Qualidade** (QA – *Quality Assurance*) é o processo geral de definição de como a qualidade de software pode ser atingida e como a organização de desenvolvimento sabe que o software tem o nível de qualidade necessário. A garantia de qualidade estabelece processos, procedimentos e padrões que conduzem a um software de qualidade.<sup>8</sup> Segundo a ISO 12207,<sup>44, 45, 46</sup> a garantia de qualidade é um processo para garantir que os processos e produtos de software, no ciclo de vida do projeto, estejam em conformidade com seus requisitos especificados e sejam aderentes aos planos estabelecidos.



Já Chrissis, Konrad e Shrum<sup>49</sup> definem garantia de qualidade como um meio planejado e sistematizado para assegurar à gerência que padrões, práticas, procedimentos e métodos definidos do processo são aplicados.

Em resumo, a garantia de qualidade assegura que um processo de desenvolvimento adequado está sendo usado, que os projetos usam padrões e procedimentos nas suas atividades, as revisões e auditorias independentes são conduzidas, uma documentação é produzida para apoiar as atividades de manutenção, uma documentação é produzida durante e não após o desenvolvimento, técnicas são usadas para controlar as mudanças, não conformidades sobre padrões e que procedimentos sejam descobertos tão logo quanto possível etc.<sup>38</sup>

Os padrões de software podem ser definidos para produtos ou processos. Como padrões de produto podem ser considerados estrutura de documentos de Engenharia e de gerenciamento, estilos de programação em Java etc. Como padrões de processo podem ser considerados processos de desenvolvimento, processos de V&V, processos de gerenciamento etc.

O **Planejamento de Qualidade** tem por objetivo selecionar e adaptar os processos, procedimentos e padrões que serão aplicados em um projeto específico.<sup>8</sup> Como resultado desta atividade, um plano de qualidade é gerado para o projeto.

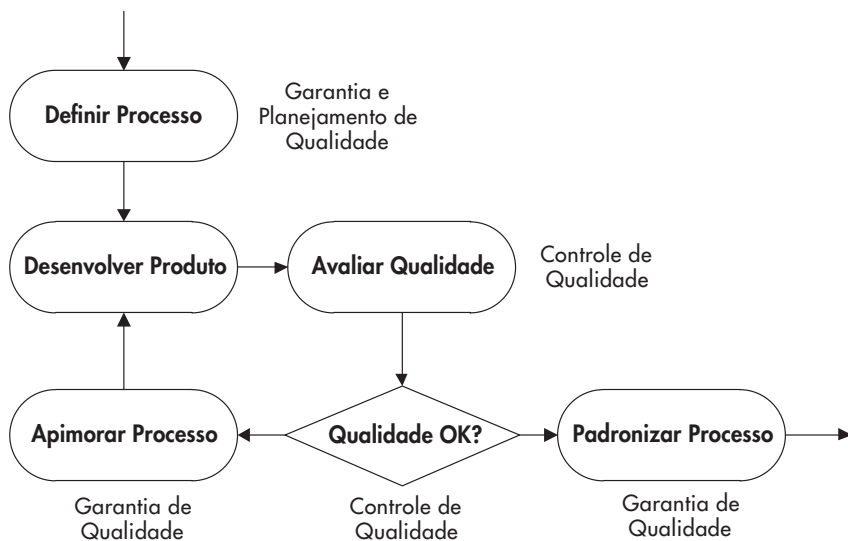
Um plano de qualidade define os meios que serão usados para assegurar que o software desenvolvido satisfaça os requisitos do usuário e é de mais alta qualidade possível dentro das restrições do projeto.<sup>55</sup> O plano identifica documentos, padrões, práticas e convenções que conduzem o projeto e como eles serão verificados e monitorados para assegurar adequação e aderência.<sup>7</sup> Incluem-se neste plano, uma descrição do produto, planos de distribuição do software, descrição de processo de desenvolvimento, metas de qualidade, análise de riscos etc.

O **Controle de Qualidade** tem por objetivo assegurar que os processos, procedimentos e padrões tenham sido seguidos pela equipe de desenvolvimento.<sup>8</sup> Incluem as revisões e auditorias de qualidade, avaliações de desempenho do projeto e medições de qualidade (número de não conformidades).

Para atingir os objetivos do gerenciamento de qualidade, uma equipe independente deve ser designada para as atividades previstas, de modo que ela tenha autonomia para levantar as eventuais não conformidades de maneira crítica e imparcial.

A Figura 8.3 apresenta um processo de Gerenciamento de Qualidade, segundo Sommerville.<sup>8</sup>

Uma norma importante para implementar gerenciamento de qualidade é a ISO 12207.<sup>44, 45, 46</sup> Esta norma possui um processo de garantia de qualidade que prevê as atividades definidas anteriormente. As atividades são a implementação do processo na qual se definem os padrões e procedimentos a serem aplicados, a garantia de qualidade do produto e do processo que trata do controle de qualidade do produto e do processo de acordo com os padrões e procedimentos definidos e, finalmente, o estabelecimento de um sistema de garantia de qualidade em que se garante que o processo esteja de acordo com outras normas, como a norma ISO 9001.<sup>53, 54</sup>



**Figura 8.3** – Gerenciamento de Qualidade (adaptado).<sup>8</sup>

Os modelos CMMI e MR-MPS também possuem processos de garantia de qualidade. (Veja mais detalhes no Capítulo 11.)

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

Como foi visto no capítulo anterior, as atividades de Gerenciamento de Configuração estão relacionadas com as atividades de garantia de qualidade. Da mesma maneira a IEEE 730-2002<sup>55</sup> indica que um dos documentos de um Plano de Garantia de Qualidade de software é o Plano de Gerenciamento de Configuração.

É comum as organizações definirem a garantia de qualidade como uma área de apoio às atividades de desenvolvimento de software. Esta área não é responsável por desenvolver software de qualidade, sendo esta tarefa de responsabilidade de todos os *stakeholders* do projeto de software. Entretanto, ela é responsável por auditar o processo e o software resultante e alertar a gerência sobre as não conformidades encontradas.

Assim, é um erro pensar que a área de garantia de qualidade é a responsável direta pela qualidade. A existência desta área não assegura que os padrões e procedimentos sejam seguidos. Se a gerência não demonstrar periodicamente seu apoio à garantia de qualidade para seguir as suas definições, ela não será efetiva. A área deve negociar a resolução das não conformidades com a gerência de projetos antes da comunicação à alta gerência.<sup>38</sup>

Mas, a alta gerência deve insistir para que as não conformidades sejam resolvidas antes de o software ser liberado para o cliente, senão, a garantia de qualidade se tornará uma atividade burocrática dispendiosa.

## EXERCÍCIOS

1. Quando o planejamento de qualidade deve ser realizado? Justifique.
2. Qual seria uma maneira adequada de abordar as não conformidades com a alta gerência?
3. Pesquise e descreva um indicador de qualidade para o gerenciamento de qualidade.
4. Descreva como o gerenciamento de projeto pode ser apoiado com o gerenciamento de qualidade.

5. Quem deve ser responsável pela qualidade de software? Justifique.
6. O papel de gerente de qualidade pode ser desempenhado pelo gerente de projeto? Justifique.
7. Sugira um critério para identificar não conformidades em processos e produtos de software.
8. Por que se diz que tendo uma área de garantia de qualidade assegura-se que o produto tem qualidade?
9. Como se poderiam introduzir as auditorias de qualidade nos projetos de software?
10. As atividades de Gerenciamento de Qualidade também devem ser auditadas? Sugira uma forma de o Gerenciamento de Qualidade ser auditado.

## SUGESTÕES DE LEITURA

BOURQUE, Pierre; DUPUIS, Robert; TRIPP, Leonard L. (Eds.). *Guide to the software engineering body of knowledge – SWEBOOK*. USA: IEEE Computer Society Press, 2004.

IEEE. {I}IEEE Std 730-2002 – IEEE Standard for software quality assurance plans. IEEE, Sep., 2002.{/I}

RUNESON, P.; ISACSSON, P. Software quality assurance – concepts and misconceptions. In: *Proceedings of the 24<sup>th</sup> Euromicro Conference*, vol. 2, pp. 853-859, 1998.

ISO. *ISO/IEC 12207 Information technology – Software life cycle processes*. ISO. 1995.

ISO. *ISO/IEC 12207 Information technology – software life cycle processes – amendment 1*. ISO. 2002.

ISO. *ISO/IEC 12207 Information technology – software life cycle processes – amendment 2*. ISO. 2004.

ISO. *ISO/IEC 12207 System and Software Engineering – Software Life Cycle Processes*. ISO. 2008.

SCHULMEYER, Gordon G.; McMANUS, James I. *Handbook of software quality assurance*. 3<sup>rd</sup> edition. New Jersey: Prentice Hall, 1999.

## Ferramentas CASE

---

**P**ara apoiar as atividades de desenvolvimento são usadas ferramentas CASE (*Computer Aided Software Engineering*). Todas as atividades referentes ao desenvolvimento de software podem ser apoiadas por alguma ferramenta de software. Inicialmente, havia apenas compiladores, ligadores e carregadores de código. Atualmente, há ferramentas de diversos tipos e necessidades, desde gerenciamento de projetos, desenvolvimento de software até documentação. Elas apoiam os profissionais na execução de suas atividades proporcionando mais qualidade e produtividade ao desenvolvimento de software.

---

### 9.1. INTRODUÇÃO ÀS FERRAMENTAS CASE

*Os processos de desenvolvimento de software exigem a aplicação de conceitos e técnicas seguindo métodos de desenvolvimento. As ferramentas CASE surgiram para apoiar as atividades de desenvolvimento, proporcionando qualidade e produtividade ao software. As ferramentas podem ser classificadas na perspectiva funcional (gerenciamento, prototipação etc) ou de processo (análise, projeto, codificação e testes). No mercado, existem muitas ferramentas disponíveis. O ArgoUML, mantido pela comunidade Tigris, o Eclipse, mantido pelo Eclipse Foundation, o Together, da empresa Borland e o Rational Rose, da empresa Rational/IBM, são exemplos de ferramentas.*

As ferramentas CASE foram introduzidas nas décadas de 1980 e 1990 e, de forma geral, são constituídas de softwares para apoiar os processos de desenvolvimento e evolução de software. Existem vários tipos, entre eles:

- Editores gráficos para desenvolvimento de modelos gráficos de sistema como parte da especificação de requisitos ou do projeto de software.
- Dicionário de dados, normalmente usado como um repositório central, para gerenciar todas as informações definidas de projeto, a fim de ter uma compreensão melhor de um projeto.
- Construtores de interfaces gráficas com o usuário, para a construção de interfaces de maneira iterativa.
- Depuradores para apoiar a detecção de defeitos de programas por execução.
- Tradutores automáticos para gerar novas versões de um programa a partir de uma linguagem antiga.

A tecnologia CASE está disponível na maioria das atividades rotineiras dos processos de software e tem elevado os níveis de qualidade e produtividade do software. Porém, as expectativas sobre os avanços dessa tecnologia desde a sua introdução não foram plenamente atendidas, em muitos aspectos, devido aos seguintes fatores:<sup>36</sup>

- Engenharia de Software é, essencialmente, uma atividade de projeto e requer pensamento criativo – isso não é prontamente automatizado.
- Engenharia de Software é uma atividade de equipe e, para projetos de grande porte, muito tempo é despendido nas interações da equipe. A tecnologia CASE não apoia realmente essas interações.

As ferramentas CASE podem ser classificadas de acordo com algumas perspectivas:<sup>36</sup>

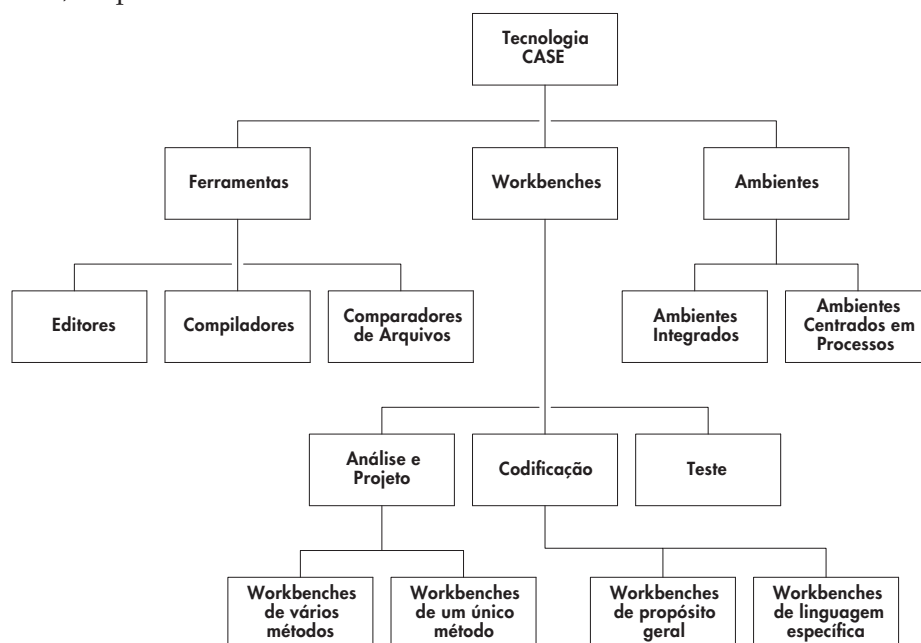
- Perspectiva funcional: Classificação de acordo com a sua função específica. Nesta perspectiva, têm-se ferramentas de planejamento, edição, gerenciamento de configuração, prototipação, compiladores, testes e depuração.
- Perspectiva do processo: Classificação de acordo com atividades de processo que são apoiadas. Nesta perspectiva, têm-se ferramentas para as atividades de análise (ferramentas de apoio a métodos), projeto (ferramentas de apoio a métodos), codificação (geradores de código) e testes (ferramentas de testes e de depuração).
- Perspectiva de integração: Classificação de acordo com sua organização em unidades integradas que apoiam uma ou mais atividades de processo.

Outra classificação possível de ferramentas CASE é baseada na abrangência de apoio ao processo de software. Existem três categorias:<sup>36</sup>

- Ferramentas: Apoiam tarefas individuais de processo como verificação de consistência de projeto, edição de texto etc.
- Bancadas (*Workbenches*): Apoiam uma fase de processo como especificação ou projeto. Normalmente, incluem uma série de ferramentas integradas.
- Ambientes: Apoiam todo ou parte substancial de um processo inteiro de software, desde a modelagem de processo até a sua execução. Normalmente, incluem vários *workbenches* integrados.

A Figura 9.1 apresenta um exemplo de classificação de ferramentas CASE.

No mercado existem muitas ferramentas disponíveis, desde as não comerciais (livres e abertas) até as comerciais. Entre as não comerciais têm-se o ArgoUML, mantido pela comunidade Tigris, o qual está disponível desde 1998 e o Eclipse, mantido pelo Eclipse Foundation, que está disponível desde 2004. Entre os comerciais têm-se o Together, da empresa Borland, disponível desde 2007 e o Rational Rose, da empresa Rational/IBM, disponível desde 1992.



**Figura 9.1** – Exemplo de classificação de tecnologia CASE.<sup>36</sup>

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

Existem muitas opções de ferramentas no mercado e, também, na internet. Porém, como toda ferramenta, ela serve para apoiar as atividades manuais proporcionando mais controle e segurança das informações e não toma decisões de projeto necessárias para atingir os objetivos. Neste sentido, uma ferramenta é passiva, ou seja, depende das decisões dos desenvolvedores.

Muitas iniciativas de implantação de ferramentas CASE em organizações não foram bem sucedidas, pois faltava o essencial. Faltava um processo sobre o qual a ferramenta deveria ser configurada. É necessário, antes de tudo, um planejamento. Esse planejamento passa pela definição de um processo de desenvolvimento de software. Incluem-se, também, a definição dos recursos humanos e materiais, além de artefatos requeridos e produzidos.

As ferramentas CASE, embora úteis, estão cada vez mais complexas e abrangentes. Uma ferramenta CASE não faz milagres e não projeta e implementa software automaticamente. No entanto, devido à sua finalidade, as ferramentas CASE embutem um conceito ou abordagem que dirige o desenvolvedor a chegar ao software de maneira organizada.

Pode-se dizer com isto, em certa medida, que uma ferramenta CASE pode afetar a criatividade dos desenvolvedores. A criatividade dos desenvolvedores é afetada na proporção inversa da capacidade de domínio de conceitos e técnicas de desenvolvimento de software por parte dos desenvolvedores e não devido às características de uma ferramenta CASE.

## EXERCÍCIOS

1. Quais são as ferramentas atualmente em uso no seu processo de desenvolvimento de software?
2. A quais classes pertencem essas ferramentas (baseie-se na classificação dada na Figura 9.1)?



3. Existem diversas ferramentas CASE disponíveis no mercado. Quais classes você considera mais relevantes. Por quê?
4. As ferramentas CASE estão se tornando cada vez mais automatizadas e complexas, como fica a criatividade dos desenvolvedores?
5. Se a sua organização fosse escolher uma ferramenta CASE, quais critérios você levaria em conta?
6. As ferramentas atuais implementam uma série de recursos de apoio ao projeto. No entanto, poucos recursos são usados pelos desenvolvedores. Por que isso ocorre?
7. Os projetos de software de grande porte são normalmente desenvolvidos em locais geograficamente distribuídos. Como as ferramentas CASE podem apoiar estes projetos se cada local pode aplicar processos e métodos diferentes?
8. Muitas organizações usam ferramentas CASE somente para documentação de projeto. Embora a documentação seja importante, qual é o erro desse enfoque?
9. Uma ferramenta CASE é configurada com processos, métodos e técnicas de Engenharia de Software. Por que não é produtivo aprender conceitos pela ferramenta?
10. É possível uma ferramenta CASE baixar o nível de produtividade e qualidade de software? Justifique.

### SUGESTÕES DE LEITURA

FISCHER, Alan S. *CASE: using software development tools*. 2<sup>nd</sup> Edition. New York: Wiley, 1991.

PFLEEGER, Shari L. *Software engineering: theory and practice*. 2<sup>nd</sup> edition. USA: Prentice Hall, 2001.

PRESSMAN, R. S. *Software Engineering – A Practitioner’s Approach*. 6<sup>th</sup>. edition. McGraw Hill, 2007.

SOMMERVILLE, Ian. *Engenharia de Software*. Tradução: Selma Shin Shimizu Melnikoff, Reginaldo Arakaki, Edilson de Andrade Barbosa. Revisão Técnica: Kechi Hiram. 8. ed. São Paulo: Pearson Addison-Wesley, 2007.

## Tecnologias

---

A disciplina de Engenharia de Software tem evoluído muito desde a sua criação em 1969. Neste tempo, processos, métodos, técnicas, ferramentas e diversas tecnologias foram desenvolvidas. Atualmente, o reuso de software, arquitetura orientada a serviços e orientação a aspectos têm sido discutidas.

Novas tecnologias ainda estão sendo pesquisadas para dar conta da crescente complexidade do software.

---

### 10.1. REUSO DE SOFTWARE

*O reuso é uma forma de aumentar a produção com a aplicação de sistemas ou componentes existentes. Isso ocorre nas disciplinas de engenharia, mas, na Engenharia de Software, o reuso sistemático de software é ainda incipiente. O reuso de software pode ocorrer em tamanhos ou granularidades diferentes, desde componentes específicos até sistemas inteiros. Uma das vantagens do reuso de software é a redução dos custos totais de produção. Uma das desvantagens é o custo de entendimento e a confiabilidade do componente a ser reusado. Dada a variedade de oportunidades de reuso, é importante realizar um bom planejamento, considerando um cronograma de desenvolvimento, ciclo de vida do software, domínio da aplicação e plataforma de execução do software, entre outros.*

O conceito de reuso não é novo. Está presente em todas as atividades do dia a dia, simplesmente porque o reuso está relacionado com resolver um problema a partir de algo (conhecimento ou componente) que é conhecido e dominado. Os sistemas são projetados por meio da composição de componentes existentes, que foram usados com sucesso em outros sistemas.

Na engenharia tradicional (elétrica, mecânica etc.) o reuso é parte do processo de engenharia. O que ocorre durante a seleção de um componente é um ajustamento e uma avaliação contínua de requisitos.<sup>56</sup>

Na Engenharia de Software o foco, em geral, se dá no desenvolvimento de software a partir do “zero”, desde o seu início, procurando satisfazer os requisitos definidos. Uma vez aceitos, os requisitos nunca são questionados e devem ser implementados completa e corretamente.<sup>56</sup>

Porém, atualmente é reconhecido que, para desenvolver um software melhor, mais rapidamente e a custos reduzidos, necessita-se adotar um processo de projeto que seja baseado no reuso sistemático de software.<sup>8</sup>

Para as organizações, o software é considerado um ativo valioso e o reuso, por sua vez, possibilita aumentar o retorno sobre investimentos em software. O reuso de software pode ser genericamente definido como o uso de conhecimento ou artefatos de engenharia de sistemas existentes para implementar novos sistemas. É uma tecnologia para melhorar a qualidade e a produtividade de software.<sup>57</sup>

Dentro desse contexto, as unidades de software reusadas podem apresentar diferentes granularidades e níveis de abstração.<sup>56</sup> Por exemplo, no reuso de aplicação de sistema, o sistema inteiro pode ser reusado, sem mudanças, em outros sistemas (reuso de software de prateleira, COTS – *Commercial of the Shelf*) ou pelo desenvolvimento de famílias de aplicações; no reuso de componentes, em que os componentes de uma aplicação, desde subsistemas a objetos simples, podem ser reusados; e no reuso de objetos e funções, em que os componentes de software que implementam um objeto único bem definido ou função podem ser reusados.<sup>8</sup>

Uma das vantagens do reuso de software é a redução de custos totais de desenvolvimento. A ideia é que poucos componentes precisariam ser

especificados, projetados, implementados e validados. Outras vantagens são: aumento de confiança, menor risco ao projeto de software, uso mais eficiente de especialistas, conformidade com padrões e entregas mais rápidas. No entanto, existem alguns problemas no reuso de software. Os maiores problemas estão relacionados aos custos de entendimento do componente a ser reusado e de testes para verificar se o componente é confiável.<sup>8</sup>

Existe uma grande variedade de oportunidades de reuso desde funções simples até sistemas completos. Entre elas, podem-se citar: padrões de projeto (*design patterns*), desenvolvimento baseado em componentes, *frameworks* de aplicação, arquitetura orientada a serviços (SOA – *Service Oriented Architecture*), linhas de produtos, integração de COTS, geradores de código etc.

Portanto, é necessário um planejamento do reuso de software. Devem ser considerados os seguintes fatores:<sup>8</sup>

- O cronograma de desenvolvimento do software.
- Se o projeto de software tem restrição de prazo, uma maneira é reusar sistemas completos ao invés de componentes individuais.
- O ciclo de vida do software esperado.
- Deve-se dar prioridade à facilidade de manutenção do software quando este tiver um ciclo de vida longo. Pode haver dificuldades no futuro se houver necessidade de mudanças nos componentes desenvolvidos ou adquiridos.
- O conhecimento, as habilidades e a experiência da equipe de desenvolvimento.
- O reuso adequado de software exige tempo. Assim, devem-se desenvolver as capacidades da equipe onde ela é mais eficiente.
- A criticidade do software e seus requisitos não funcionais.
- Os sistemas críticos que devem passar por certificações de órgãos reguladores terão que apresentar evidências sobre a confiabilidade do software. Se o software tem requisitos de desempenho restritivos, os geradores de código podem não reusar o software eficientemente.
- O domínio da aplicação.
- Dependendo do domínio da aplicação, existem vários sistemas completos que, por meio de configurações, podem se tornar muito úteis. É o caso, por exemplo, de sistemas de informações gerenciais do tipo ERP (*Enterprise Resource Planning*).

- A plataforma de execução para o software.
- Deve-se considerar o tipo de plataforma a ser usada para executar o software. Por exemplo, se for Microsoft, considere os componentes do tipo COM/Active X. Assim, o software poderá ser reusado na mesma plataforma para o qual foi projetado.

Um modelo de processo de reuso pode ser visto na Figura 10.1. O modelo é inspirado em uma linha de montagem da indústria de manufatura. Basicamente, existem duas áreas concorrentes: Engenharia de Domínio e Engenharia de Aplicação. A Engenharia de Domínio estabelece um conjunto de componentes ou artefatos de software e modelos de domínio que têm aplicabilidade em sistemas existentes e futuros, dentro de um domínio de aplicação. Os componentes ou artefatos são armazenados em um repositório, que podem ser reusados pela área de Engenharia de Aplicação.<sup>3</sup> A área de Engenharia de Aplicação recebe os requisitos do usuário e desenvolve novos sistemas de software a partir do reuso dos componentes ou artefatos e modelos de domínio disponibilizados pela Engenharia de Domínio.

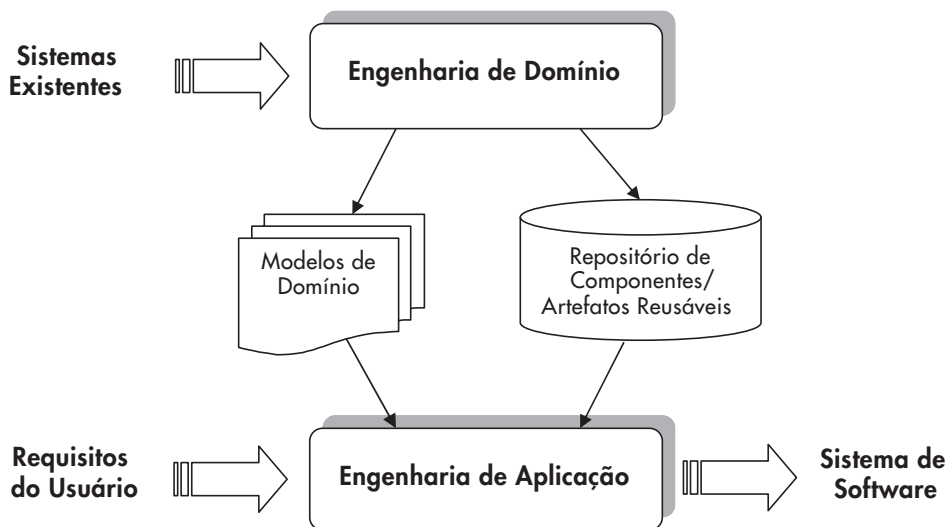


Figura 10.1 – Modelo de processo de reuso.

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

O reuso de software não é tão trivial. Nem sempre o sistema de software disponível pode ser totalmente aproveitado. Isso significa que há a necessidade de se adaptar o software para ser reusado. Deve-se também analisar a capacidade de reuso, envolvendo fatores técnicos e humanos.

Em situações mais gerais, as adaptações necessárias podem ser muito trabalhosas e podem ser, no fim das contas, muito custosas. Cabe então uma análise dos benefícios. O item a ser reusado também deve ter uma granularidade alta. Muitas vezes, itens de granularidade mais baixa, sendo mais específicos, possuem naturalmente um potencial maior de reuso, porém não proporcionam grande ganho de produtividade.

Segundo Frakes e Isoda,<sup>57</sup> muitas organizações tem procurado maximizar o reuso de software, tentando sistematizar o reuso de artefatos de maior granularidade. A sistematização do reuso pode ser uma vantagem competitiva para as organizações. Porém, se a sistematização não for bem-sucedida, perdem-se preciosos tempo e recursos e a gerência pode ter dúvidas em relação a uma nova tentativa. Entretanto, se os seus concorrentes forem bem-sucedidos, uma organização pode perder sua participação no mercado e, possivelmente, todo o mercado.

Uma importante pesquisa foi feita no Brasil entre os anos de 2004 e 2006, sobre o reuso de software no cenário industrial brasileiro. O objetivo da pesquisa foi tentar relacionar as características de organizações de desenvolvimento de software com a adoção de reuso, a fim de determinar fatores decisivos no sucesso de reuso. foram coletados dados de 57 empresas (pequenas, médias e grandes) brasileiras, sendo a maioria da região nordeste, das quais 53% obtiveram sucesso.<sup>58</sup>

Levando-se em conta que os softwares estão cada vez maiores e mais complexos, é de se supor que o reuso de software seja o caminho natural para o cumprimento de prazos e custos de desenvolvimento. Porém, o reuso efetivo de software em grande escala ainda levará algum tempo.

Uma abordagem muito interessante nesta direção são os sistemas orientados a serviços baseados no conceito de Arquitetura Orientada a Serviços (SOA – *Service Oriented Architecture*). (Veja mais detalhes na próxima seção).

## EXERCÍCIOS

1. Existem inúmeras formas de reuso, qual é a mais usada nos seus projetos de software?
2. Identifique cinco pontos fortes e fracos do reuso de software.
3. Por que o reuso de software melhora a qualidade do produto?
4. Quais são os principais obstáculos de um reuso efetivo de software?
5. Quais são os riscos de um projeto de software com reuso?
6. Identifique características de domínio importantes de um portal de comércio eletrônico.
7. O reuso de componentes de código é a forma mais comum em projetos de software. Quais são as dificuldades de usar componentes de maior granularidade e níveis de abstração?
8. Em que condições é melhor desenvolver um novo componente do que usar um existente?
9. Qual é o perfil adequado de profissional para a área de Engenharia de Domínio?
10. O conceito de reuso de software surgiu como um novo paradigma de desenvolvimento de software há algum tempo. Em sua opinião, por que o reuso ainda não é plenamente adotado nas organizações?

## SUGESTÕES DE LEITURA

CARD, David N.; COMER, Edward R. Why do so many reuse programs fail? *IEEE Software*, vol. 11, issue 5, pp. 114-115, Sep., 1994.

FRAKES, William B.; FOX, Christopher J. *Sixteen questions about software reuse. Communications of ACM*, vol. 38, issue 6, pp. 75-87, Jun., 1995.

FRAKES, William B.; ISODA, Sadahiro. Success factors of systematic reuse. *IEEE Software*, vol. 11., issue 5, pp. 14-19, Sep., 1994.

FRAKES, William.; TERRY, Carol. Software reuse: metrics and models. *ACM Computing Surveys*, vol. 28, issue. 2, pp. 415-435, Jun., 1996.

LIM, Wayne C. Effects of reuse on quality, productivity and economics. *IEEE Software*, vol. 11, issue 5, pp. 23-30, Sept., 1994.

LUCRÉDIO, Daniel; BRITO, Kellyton S.; ALVARO, Alexandre; GARCIA, Vinicius C.; ALMEIDA, Eduardo S.; FORTES, Renata P. M.; MEIRA, Silvio L. Software reuse: the brazilian industry scenario. *The Journal of Systems and Software*, vol. 81, issue 6, pp. 996-1013, Jun., 2008.

MEYER, Bertrand. Reusability: the case for object-oriented design. *IEEE Software*, vol. 4, issue 2, Mar., 1987.

PRIETO-DIAZ, Ruben. *Reuse as a new paradigm for software development. systematic reuse: issues in initiating and improving a reuse program*. In: Proceedings of the International Workshop on Systematic Reuse. London, Jan., 1996.

## 10.2. ARQUITETURA ORIENTADA A SERVIÇOS

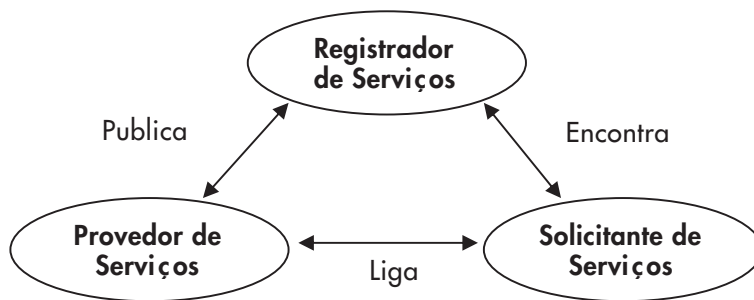
A Arquitetura Orientada a Serviços (SOA – Service Oriented Architecture) é um estilo de arquitetura usado para construir sistemas de software baseados no acesso a informações diversas disponíveis em vários computadores ligados na internet. Com a SOA, uma organização pode publicar suas informações e torná-las disponíveis para outros computadores. Neste contexto, surgem as figuras do provedor, registrador e solicitador de serviços. Um serviço pode ser definido como um componente de software reusável, fracamente acoplado, que encapsula funcionalidade discreta e que pode ser distribuído e acessado através de um programa. Os benefícios da SOA são: serviços podem ser providos localmente ou terceirizados para provedores externos, serviços são independentes de linguagem, investimentos em sistemas legados podem ser preservados e a computação interorganizacional é facilitada por meio da troca simplificada de informações.

A internet tornou possível o acesso de computadores de clientes a diversos servidores remotos intra e interorganizações. Há algum tempo, as informações disponíveis eram convertidas em linguagem HTML e, usando-se um navegador (browser), era possível ter acesso a essas informações. Porém, o acesso direto às informações não era adequado e prático.



A Arquitetura Orientada a Serviços (SOA) é um paradigma de construção e integração de software que estrutura aplicações em elementos modulares chamados serviços. O serviço, a unidade fundamental de uma arquitetura SOA, é um elemento computacional que tem como propósito desempenhar uma função específica e que pode ser utilizado por um cliente.<sup>59</sup> As aplicações usarão estes serviços por composição ou simplesmente colocando-os juntos.<sup>60</sup>

O fornecimento de serviços é independente da aplicação que usa este serviço. Uma arquitetura de aplicações baseada em serviços pode ser dividida em três partes principais: provedor, solicitante (cliente) e registrador de serviços,<sup>60</sup> conforme a Figura 10.2. As aplicações podem ser construídas de acordo com a arquitetura conceitual de um sistema orientado a serviços.



**Figura 10.2** – Modelo de arquitetura de aplicações SOA.<sup>60</sup>

Os provedores projetam e implementam os serviços especializados para uma gama de consumidores. Em seguida, publicam os serviços em um registrador de acesso geral. Os solicitantes buscam no registrador a especificação de serviço desejado e localizam o provedor. O provedor então passa a se comunicar disponibilizando o serviço ao solicitante.

Um serviço pode ser definido como um componente de software reusável fracamente acoplado que encapsula funcionalidade discreta e este pode ser distribuído e acessado através de um programa. Existem algumas tecnologias que podem ser usadas para a implementação de SOA: Web Service, Corba, RMI, DCOM, REST etc.

No caso de Web Service, um serviço que é acessado usando protocolos padrões da internet e baseados em XML (padrão para troca de informações em ambiente distribuído), o que o torna independente da linguagem usada na sua implementação.<sup>8</sup>

As organizações que desejam tornar acessíveis suas informações a outros computadores podem fazê-lo definindo e publicando uma interface Web Service. Pode-se dizer que Web Service é uma representação padronizada de alguns recursos e de informações que podem ser usadas por outros computadores.

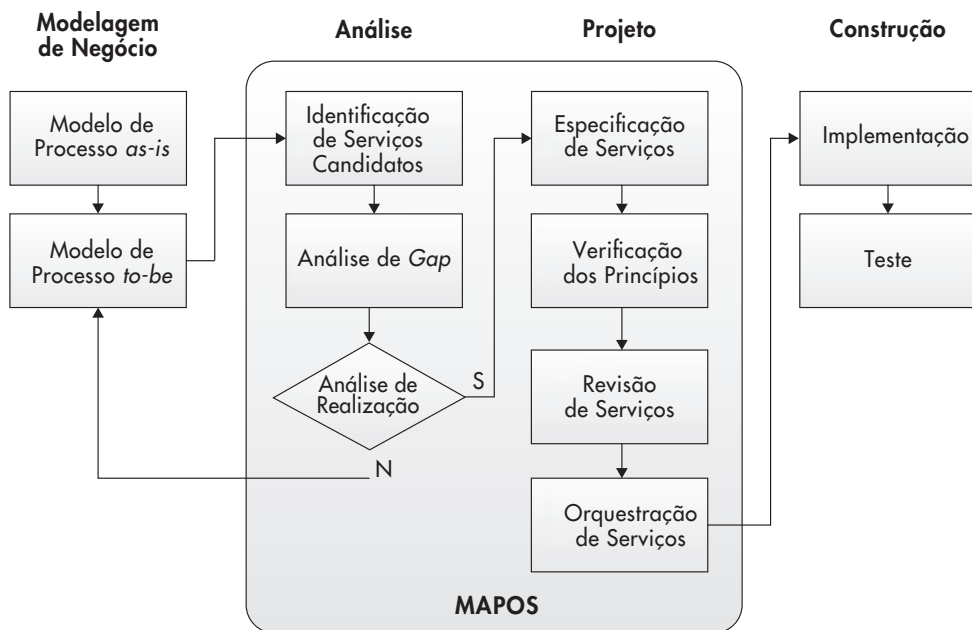
Um exemplo de aplicação de Web Service é o serviço de informações de crédito da Serasa disponibilizado aos bancos e às instituições financeiras através da internet. Nesse caso, um banco pode decidir por uma proposta de financiamento de um cliente após consultar as informações disponibilizadas pela Serasa. A Web Service cuida da verificação das informações do cliente e retorna a situação de crédito deste cliente para o banco.

Outra forma importante de aplicação da SOA é prover acesso à funcionalidade embutida em sistemas legados. Os sistemas legados oferecem funcionalidade extensiva e isso pode reduzir o custo de implementação de serviços. As aplicações externas podem acessar esta funcionalidade através de interfaces de serviço. Ou seja, os sistemas legados podem ser “encapsulados” em serviços, tornando-se disponíveis ao acesso por outros sistemas, independentemente da tecnologia empregada.

A SOA traz benefícios tais como: serviços podem ser providos localmente ou terceirizados para provedores externos; serviços são independentes de linguagem; investimentos em sistemas legados podem ser preservados; e computação interorganizacional é facilitada por meio da troca simplificada de informações.

Porém, um aspecto importante relacionado à implementação da SOA é estabelecer um método para desenvolver os serviços que satisfaçam as metas do negócio e agreguem valor a ele. É necessário garantir que os serviços desenvolvidos na organização sejam eficientes, além de operar de acordo com os requisitos funcionais, de desempenho, de qualidade e eficazes, atendendo às necessidades críticas de negócio.<sup>59</sup>

Um método interessante foi proposto por Fugita.<sup>59</sup> A Figura 10.3 ilustra o método MAPOS proposto e as suas fases.



**Figura 10.3** – MAPOS: Método de Análise e Projeto Orientado a Serviços.<sup>59</sup>

O método MAPOS é constituído de duas fases: Análise e Projeto. O método é precedido pela atividade de Modelagem de Negócio e sucedido pela atividade de Construção de Serviços.<sup>59</sup>

Na atividade de Modelagem de Negócio busca-se um entendimento sobre as necessidades de negócio e levantam-se os requisitos.

Na fase de Análise, são identificados serviços candidatos a partir dos modelos de negócio fornecidos. Ainda nesta fase, decide-se como cada serviço será realizado, analisando-se possibilidades de reuso para cada um, chegando a um conjunto de serviços finais.

Definidos os serviços, inicia-se a fase de Projeto. Nesta fase, criam-se especificações de serviços que satisfaçam aos requisitos do negócio e sejam aderentes aos princípios de orientação a serviços. Como produtos desta fase,

tem-se a orquestração (composição) de serviços e especificações técnicas de serviços que servem como base para a atividade de Construção de Serviços.

Após a Análise e o Projeto, a próxima atividade é a de Construção. Nesta atividade os serviços são implementados, integrados e testados, prontos para serem disponibilizados.

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

O crescente aprimoramento dos negócios para as organizações manterem-se competitivas requer uma abordagem rápida e eficiente de atualização dos recursos computacionais que preserve investimentos.

O alinhamento dos negócios com recursos computacionais e vice-versa é crucial para este objetivo. SOA é uma abordagem que preenche uma lacuna deste alinhamento. O conceito de serviços pode ser entendido como uma camada entre as atividades de negócio e a infraestrutura computacional da organização. Uma característica importante é que a SOA torna mais fácil a configuração de aplicações a partir da composição de serviços existentes.

Há que se ressaltar, porém, que existem muitas aplicações na indústria de software baseadas na tecnologia Web Service ditas serem SOA. Por exemplo, a disponibilização de uma base de informações através da internet, em si, não constitui SOA. Ela é mais do que uma tecnologia, é uma estratégia de negócio baseada em uma arquitetura de serviços distribuídos de baixo acoplamento e alta granularidade.

Para obter uma boa implementação de SOA, necessita-se de um método que leve em conta os requisitos de negócio até chegar aos serviços propriamente ditos. Existem dois desafios importantes nos desenvolvimentos de SOA: um relacionado ao teste e outro à proteção de aplicações de SOA.

Por exemplo, se a aplicação de SOA é uma composição de serviços, como se pode garantir que as instâncias destes serviços estejam disponíveis para os desenvolvedores testá-las? Além disso, como garantir infraestrutura necessária para os testes em ambientes distribuídos? Com relação à proteção:

Como garantir que os mecanismos de proteção sejam eficazes uma vez que as aplicações estariam disponíveis para uma variedade de outras aplicações?

## EXERCÍCIOS

1. O que é um serviço? Dê um exemplo.
2. Sendo os serviços reusáveis, confundem-se com componentes de software. Quais são as diferenças entre estes conceitos?
3. Quais são, na sua opinião, os tipos de aplicações onde a abordagem de SOA não seja adequada?
4. Pesquise e descreva uma aplicação que seja de SOA. Justifique.
5. Quais são as razões estratégicas de negócio para implementar a SOA?
6. Que tipo de sistema legado pode ter sua funcionalidade disponibilizada em uma aplicação de SOA? Dê um exemplo.
7. Qual é a diferença entre um desenvolvimento de aplicações de SOA e outro no modo tradicional?
8. O conceito de SOA traz muitos benefícios, porém traz também desafios a serem enfrentados. Descreva como realizar testes em aplicações de SOA.
9. Pesquise métodos de desenvolvimento de aplicações de SOA. Compare-os em relação aos requisitos de negócio.
10. Um arquiteto de software tradicional tem conhecimentos para trabalhar com aplicações de SOA? Justifique.

## SUGESTÕES DE LEITURA

ERL, Thomas. *Service-oriented architecture: concepts, technology and design*. USA: Prentice Hall, 2005.

ERL, Thomas. *SOA principles of service design*. USA: Prentice Hall, 2007.

FUGITA, Henrique S. *MAPOS: método de análise e projeto orientado a serviços*. Escola Politécnica da USP, 2009, 175p. Dissertação de Mestrado em Sistemas Digitais, São Paulo, 2009.

HUHNS, Michael. N.; SINGH, Munindar P. *Service-oriented computing: key concepts and principles*. *IEEE Internet Computing*, vol. 9. issue. 1, pp. 75-81, Jan./Feb., 2005.

### 10.3. ORIENTAÇÃO A ASPECTOS

*A Orientação a Aspectos é um novo paradigma de desenvolvimento de software que foi criada para preencher uma lacuna onde as abordagens Estruturada e Orientada a Objetos não atendem de forma adequada. Esta lacuna é o tratamento de interesses transversais dos stakeholders ligados aos requisitos não funcionais que devem ser considerados em um sistema como um todo. Os interesses transversais causam os fenômenos conhecidos como entrelaçamento e espalhamento de código que dificultam o reuso e a manutenção de software. A Orientação a Aspectos fundamenta-se em dois princípios que são a separação de interesses e a modularização. O aspecto é um módulo de programa que é composto no programa alvo em pontos específicos.*

Todo desenvolvimento de software começa com a atividade de Análise. Na Análise de requisitos deve-se levar em conta a existência de diversos stakeholders, no qual cada um tem uma perspectiva diferente sobre o problema ou sobre soluções potenciais. Essas perspectivas refletem os diferentes interesses dos stakeholders, ou as visões alternativas que esses stakeholders têm sobre a mesma área de interesse. Portanto, essas perspectivas frequentemente se sobrepõem e conflitam umas com as outras, mas são toleradas, já que no início da Análise o objetivo é entender o problema e elicitar os requisitos, e não a resolução de conflitos.<sup>61</sup>

A Engenharia de Software fundamenta-se basicamente em dois princípios: a separação de interesses<sup>62</sup> e a modularização.<sup>63</sup> Os princípios são baseados na estratégia “dividir e conquistar” usada em outras disciplinas, não somente na engenharia. Todo problema complexo pode ser separado em problemas menores e cada módulo resolve um único problema. Na separação de interesses, o software deve ser separado em características (ou comportamentos) distintas com mínimo de sobreposição. Na modularização, cada módulo é o resultado da decomposição de um sistema. Um módulo embute uma responsabilidade de uma parte do software.

Esses princípios estão presentes na abordagem Estruturada com funções, na abordagem Orientada a Objetos com objetos, na arquitetura SOA com

serviços, no projeto de arquitetura com *design patterns*,<sup>64</sup> no desenvolvimento com reuso com componentes e na abordagem Orientada a aspectos com aspectos.

No início da Análise, os interesses dos *stakeholders* são transformados em requisitos funcionais e não funcionais. Requisitos funcionais são declarações de alguma função ou característica que deve ser implementada em um sistema. Requisitos não funcionais são declarações de uma restrição ou comportamento esperado que se aplica a todo o sistema.<sup>8</sup>

Assim, a parte do sistema relacionada aos requisitos funcionais é tradicionalmente direcionada à modularização de software, onde cada módulo é responsável por uma função importante e distinta do sistema. A parte do sistema relacionada aos requisitos não funcionais, por se aplicarem a todo sistema, normalmente acabam aparecendo em muitos módulos. Isso tem como consequência o surgimento de dois fenômenos: Entrelaçamento (*tangling*) e Espalhamento (*scattering*). Em outras palavras, os interesses dos *stakeholders* ficam entrelaçados dentro dos módulos e espalhados entre os módulos.

Os interesses podem ser divididos em interesses centrais (requisitos funcionais) e interesses transversais (requisitos não funcionais). Os interesses transversais causam o espalhamento, enquanto o entrelaçamento é uma consequência desse espalhamento.

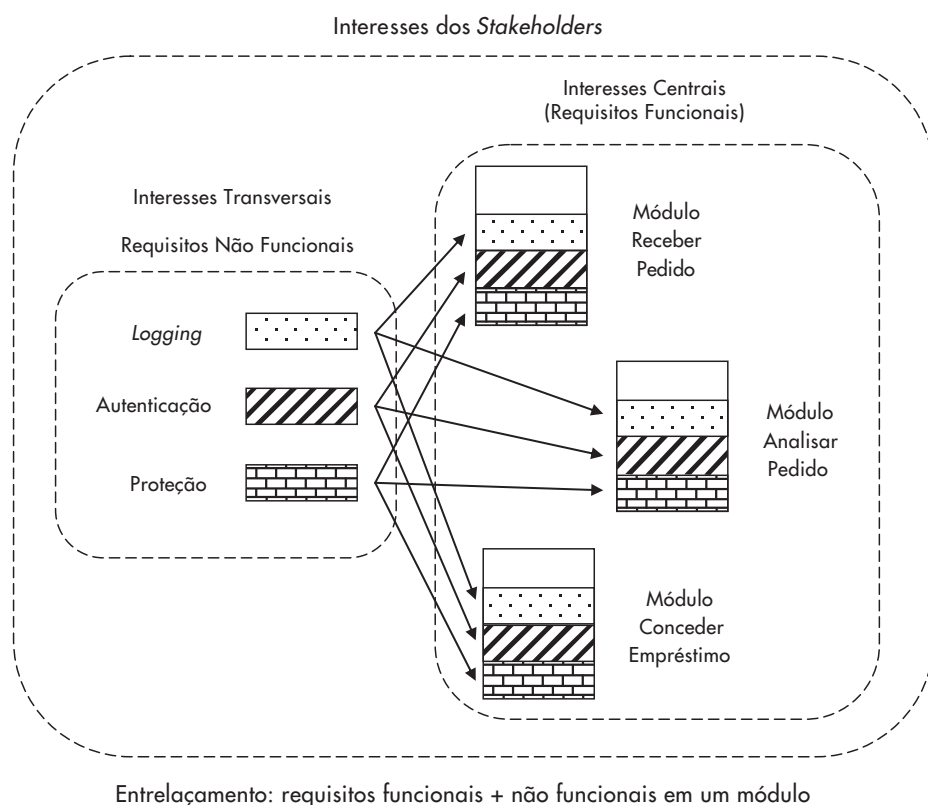
Por exemplo, suponha que uma aplicação financeira tenha como interesses centrais (requisitos funcionais) Receber Pedido, Analisar Pedido e Conceder Empréstimo e interesses transversais (requisitos não funcionais) de *Logging*, Autenticação e Proteção. Por simplicidade, assume-se que cada requisito funcional é implementado por um módulo correspondente. A Figura 10.4 ilustra como estes requisitos são normalmente implementados em módulos.

Os fenômenos de entrelaçamento e espalhamento trazem como consequência, dificuldades no reuso e na manutenção destes módulos.

Na abordagem Orientada a Objetos, os módulos são classes, objetos ou componentes. Para definir o que é aspecto, pode-se fazer uma

comparação com componente. Segundo Kiczales et al.,<sup>65</sup> um componente é fruto da decomposição funcional de sistema e pode ser encapsulado e facilmente acessado e composto (por exemplo, serviço). Um aspecto, por outro lado, não é fruto dessa decomposição e afeta o desempenho e a semântica destes componentes em termos sistêmicos (por exemplo, tratamento de exceções).

O aspecto tratamento de exceções pode ser acionado sempre que alguma condição de erro de código deve ser tratada. O tratamento de exceções pode ser embutido automaticamente dentro de um programa que necessite deste tratamento. O aspecto pode ser reusado onde for requerido e mantido de forma independente e com mais facilidade.



**Figura 10.4** – Fenômeno de entrelaçamento e espalhamento.

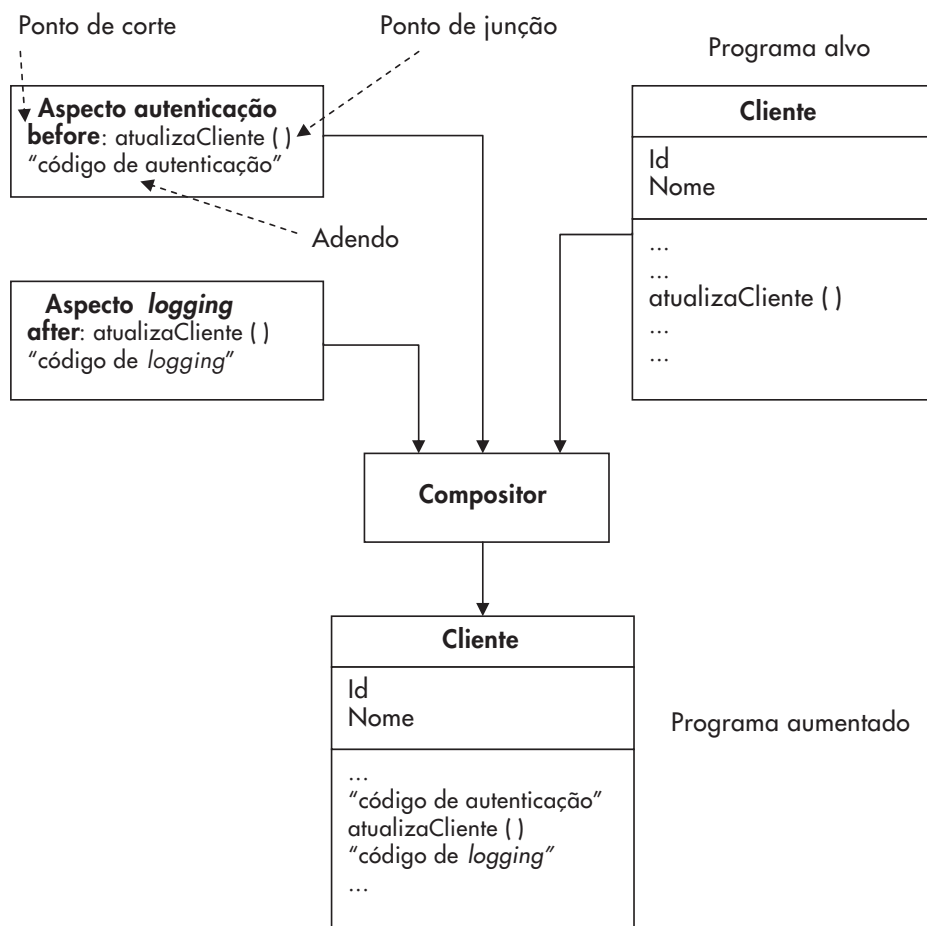


Um aspecto é uma declaração de módulo de programa que define pontos de corte e adendos.<sup>66</sup> Um ponto de corte identifica os pontos de junção no programa alvo onde um adendo deve ser executado. Os adendos fornecem um novo comportamento ao programa alvo nos pontos de junção. Um ponto de junção pode ser chamada (*call*) para método, execução (*execution*) de método, iniciação (*initialization*) de classe ou objeto, dados (*data*) em que um campo é acessado ou atualizado, tratadores (*handler*) para tratamento de exceções etc.

Um ponto de corte define quando um adendo pode ser executado no programa alvo. Um adendo pode ser executado antes (*before*), depois (*after*) ou em torno (*around*) do ponto de junção no programa alvo.

Estes adendos são incorporados nos pontos de junção por meio de compositores da linguagem de programação. Um compositor (*weaver*) gera um programa executável orientado a aspectos de duas maneiras. No modo estático, os adendos são incorporados no programa alvo em tempo de compilação e, no modo dinâmico, em tempo de execução. A Figura 10.5 apresenta uma composição de aspectos em um programa alvo (classe) para um programa aumentado (classe resultante).

Para desenvolver software neste novo paradigma, existem linguagens de programação orientadas a aspectos como AspectJ, que é uma extensão da linguagem Orientada a Objetos Java.<sup>65, 67, 68</sup> O *release* AspectJ 1.6.12.M1, de 07 de junho de 2011 é a última versão do AspectJ 6.<sup>69</sup>



**Figura 10.5** – Composição de aspectos em uma classe (adaptado).<sup>8</sup>

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

A abordagem Orientada a Aspectos é centrada no tratamento de interesses transversais. Da mesma forma que nas abordagens tradicionais existe também a Engenharia de Software Orientada a Aspectos (AOSE – *Aspect-Oriented Software Engineering*) que é uma disciplina para desenvolvimento de software que foca na separação de interesses. Ela se propõe

a resolver problemas de entrelaçamento e espalhamento e tornar os programas mais fáceis de manter e reusar.<sup>8</sup>

Como foi discutida, a abordagem Orientada a Aspectos permite o melhor tratamento de requisitos não funcionais que as abordagens tradicionais não tratam de maneira adequada.

Porém, esta abordagem ainda precisa evoluir, pois existem problemas de verificação (inspeção de código) e testes de programas orientados a aspectos<sup>8,66</sup> Um dos fatores é que os aspectos podem mudar o comportamento do programa alvo dificultando a inspeção de código e a cobertura de testes. Na inspeção de código, ler sequencialmente os caminhos de fluxo de controle de código é difícil e os testes realizados no programa alvo podem não ser mais úteis para o programa aumentado devido à incorporação de aspectos.

## EXERCÍCIOS

1. Descreva com suas próprias palavras, o que é separação de interesses e qual é a sua importância.
2. O que são interesses centrais? Dê alguns exemplos.
3. O que são interesses transversais? Dê alguns exemplos.
4. Quais são as vantagens e desvantagens da abordagem Orientada a Aspectos?
5. Em que sentido a abordagem Orientada a Aspectos complementa a Orientada a Objetos? Justifique.
6. Dê um exemplo de interesse transversal que causa os fenômenos de entrelaçamento e espalhamento.
7. O que é aspecto e sua relação com classe da abordagem Orientada a Objetos?
8. Descreva como funciona o compositor de aspectos. Dê um exemplo na linguagem AspectJ.
9. Os aspectos podem ser compostos dinamicamente no programa alvo. Como a inspeção de código pode ser realizada?
10. Pesquise ferramentas para apoiar o uso da abordagem Orientada a Aspectos. Faça um resumo de uma delas.

## SUGESTÕES DE LEITURA

JACOBSON, Ivar.; NG, Pan-Wey. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley – Pearson Education, Inc. USA, 2005.

KATZ, Shmuel. *A Survey of Verification and Static Analysis for Aspects*. AOSD-Europe-Technion-1 – M8.1. Jul., 2005.

KICZALES, Gregor; HILSDALE, Erik.; HUGUNIN, Jim.; KERSTEN, Mik; PALM, Jeffrey; GRISWOLD, William G. *An Overview of AspectJ*. In: Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP). Springer, 2001a.

KICZALES, Gregor; HILSDALE, Erik.; HUGUNIN, Jim.; KERSTEN, Mik; PALM, Jeffrey; GRISWOLD, William G. *Getting Started with AspectJ*. Communications of the ACM. vol. 44, issue 10, pp. 59-65, Oct., 2001b.

KICZALES, Gregor; LAMPING, John; MENDHEKAR, Anurag; MAEDA, Chris; LOPES, Cristina V.; LOINGTIER, Jean-Marc; IRWIN, John *Aspect-Oriented Programming*. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag LNCS 1241. Jun., 1997.

## Modelos de maturidade

---

Hoje em dia, as organizações de software precisam melhorar continuamente os seus processos para fazer frente aos desafios crescentes. Para isto, no fim da década de 1980, baseada na estratégia de TQM<sup>4,5</sup> surgiu o conceito de maturidade de processos obtida a partir de melhoria contínua de processos. Os modelos de maturidade tiveram como precursor o modelo CMM (*Capability Maturity Model*) do SEI (*Software Engineering Institute*) da Carnegie Mellon University dos Estados Unidos no início da década de 1990 que permite às organizações obterem qualidade e produtividade de software e adotarem estratégias de melhoria contínua dos seus processos. Atualmente existem o CMMI (*Capability Maturity Model Integration*) dos Estados Unidos e MR-MPS (Modelo de Referência) do Programa MPS-BR do Brasil, como os mais representativos.

---

### 11.1. CMMI

O CMMI (*Capability Maturity Model Integration*) é um modelo maturidade de melhoria contínua de processo para desenvolvimento de produtos e serviços, sob responsabilidade do SEI (*Software Engineering Institute*), da Carnegie Mellon University dos Estado Unidos. O CMMI é uma evolução do modelo CMM do início da década de 1990 e consiste em uma organização de boas práticas que cobrem as atividades de gerenciamento de projeto, gerenciamento de processo, engenharia de sistemas, engenharia de hardware,

*engenharia de software e outras atividades de apoio. As organizações podem se submeter às certificações oficiais do SEI que certificam o nível de maturidade da organização. É um modelo que tem um reconhecimento mundial, e no Brasil, há muitas empresas com estas certificações.*

O CMMI (Capability Maturity Model Integration) consiste em uma organização de boas práticas que atendem às atividades de desenvolvimento e manutenção de um ciclo de vida do produto desde a concepção até a entrega e manutenção.<sup>49, 50</sup>

O objetivo do CMMI é prover um framework comum que possibilite a integração futura de outros modelos específicos do CMMI. Uma das premissas é que todos os produtos devem ser compatíveis e consistentes com a norma ISO/IEC 15504.<sup>73</sup> Assim, o CMMI propõe-se a fornecer um guia para melhoria dos processos organizacionais e da sua capacidade de gerenciar o desenvolvimento, aquisição e a manutenção de produtos e serviços.

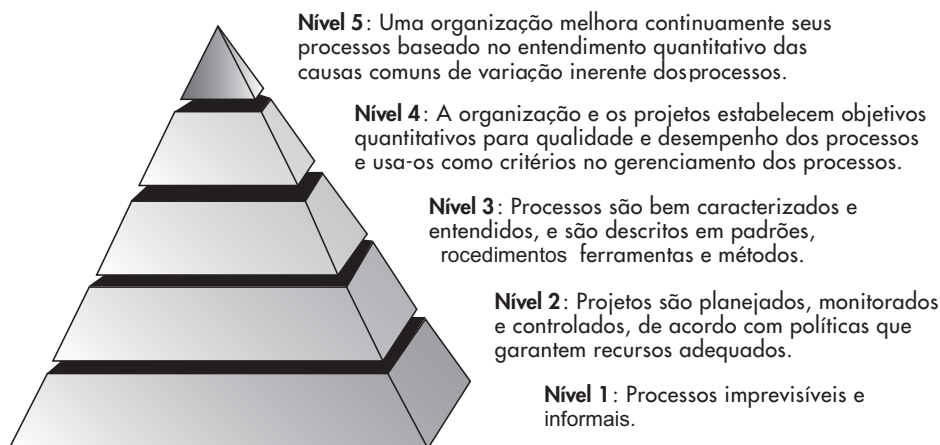
O CMMI está estruturado em três constelações: a constelação CMMI-DEV (desenvolvimento),<sup>49, 50</sup> que cobre as atividades de desenvolvimento e manutenção aplicadas a produtos e serviços; a constelação CMMI-ACQ (aquisição)<sup>70</sup> que cobre as atividades de aquisição de recursos junto aos fornecedores; e a constelação CMMI-SRV (serviço)<sup>71</sup>, que cobre as atividades de definição, entrega e gerenciamento de serviços.

O CMMI permite que uma organização defina uma abordagem de melhoria contínua de processo e avaliações usando duas representações diferentes: por estágios e contínua.

A representação por estágios provê uma sequência de melhorias da organização, começando com práticas básicas de gerenciamento e prosseguindo através de níveis sucessivos predefinidos, onde um nível serve de base para o próximo nível. Esta é a representação mais usada e reconhecida nas organizações de software.

O CMMI-DEV por estágios possui cinco níveis de maturidade. Cada nível de maturidade é um patamar bem definido evolucionário de um

caminho para tornar uma organização madura, sendo uma camada a base para melhoria contínua de processos. A Figura 11.1 apresenta os níveis de maturidade do CMMI-DEV por estágios, enfatizando os seus focos de atuação na organização.



**Figura 11.1** – Níveis de maturidade do CMMI-DEV (por estágios).

A Tabela 11.1 apresenta os níveis de maturidade e as áreas de processo (referem-se a um conjunto de boas práticas de uma disciplina) do modelo CMMI-DEV relacionadas a cada um dos níveis. Não existem áreas de processo para o nível Inicial.

O CMMI-DEV na representação contínua permite escolher a ordem de melhoria, ou seja, escolher os processos que melhor atendam aos objetivos de negócio e minimize as áreas de risco da organização. A Figura 11.2 apresenta os quatro níveis de capacidade do CMMI-DEV contínua. Os níveis de capacidade são quatro:

- **Nível 0 (Incompleto):** O processo não é executado ou parcialmente executado.
- **Nível 1 (Executado):** O processo realiza o trabalho necessário para obter os produtos e suas metas são atingidas.
- **Nível 2 (Gerenciado):** O processo é planejado e executado de acordo com as políticas, monitorado, controlado e revisado.

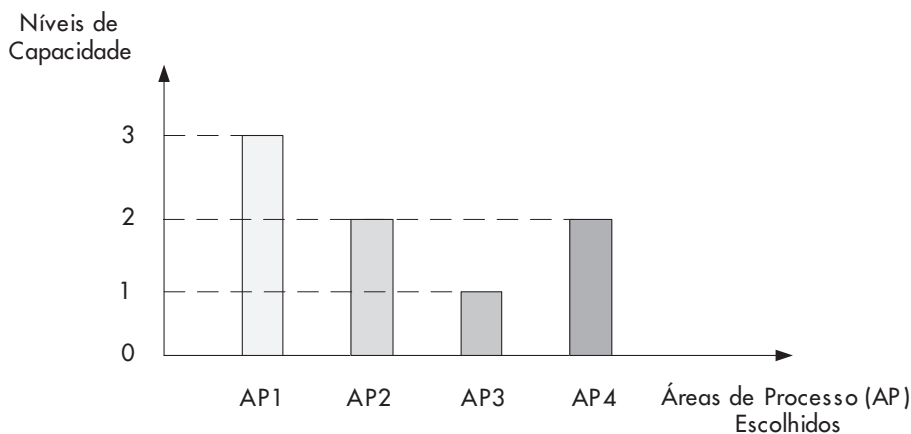
- **Nível 3 (Definido):** O processo é gerenciado e customizado a partir de um conjunto de processos padrões, de acordo com diretrizes de customização da organização.

As duas representações fornecem caminhos para implementar melhoria de processos para atingir os objetivos da organização. Em termos de avaliações e melhoria de processos, ambas oferecem essencialmente resultados equivalentes. Ambas possuem o mesmo conteúdo, mas são organizados de modos diferentes.<sup>49</sup>

**Tabela 11.1** – Áreas de processo relacionadas a cada um dos níveis de maturidade do modelo CMMI-DEV.<sup>49</sup>

NÍVEIS	ÁREAS DE PROCESSO
5 – Em Otimização	Análise Causal e Resolução – CAR Gerenciamento de Desempenho Organizacional – OPM
4 – Quantitativamente Gerenciado	Desempenho de Processo Organizacional – OPP Gerenciamento de Projeto Quantitativo – QPM
3 – Definido	Foco de Processo Organizacional – OPF Definição de Processo Organizacional – OPD Treinamento Organizacional – OT Gerenciamento de Projeto Integrado – IPM Gerenciamento de Riscos – RSKM Desenvolvimento de Requisitos – RD Solução Técnica – TS Integração de Produto – PI Verificação – VER Validação – VAL Análise de Decisão e Resolução – DAR
2 – Gerenciado	Gerenciamento de Requisitos – REQM Planejamento de Projeto – PP Monitoração e Controle de Projeto – PMC Gerenciamento de Acordo com Fornecedor – SAM Garantia de Qualidade de Processo e Produto – PPQA Gerenciamento de Configuração – CM Medição e Análise – MA
1 – Inicial	<b>Não há áreas de processo definidas.</b>

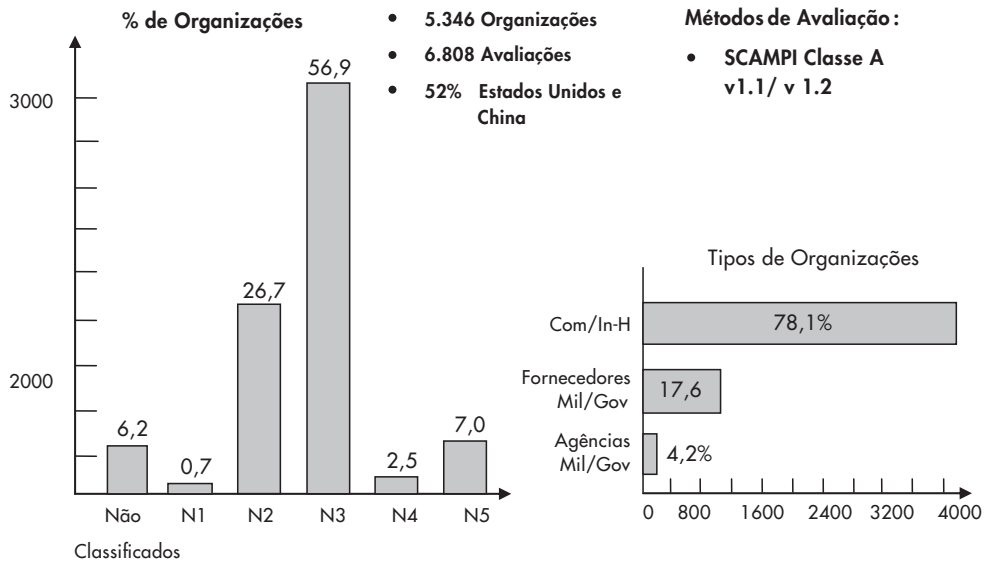




**Figura 11.2** – Níveis de capacidade do CMMI (contínua).

A Figura 11.3 apresenta o perfil das organizações com certificações CMMI (CMMI v1.1 e CMMI-DEV v1.2). Ela representa as organizações que obtiveram níveis de maturidade durante o período de abril de 2002 a junho de 2010. As organizações foram certificadas nos seus respectivos níveis pelo método SCAMPI Classe v1.1/v 1.2 do próprio SEI.

Foram realizadas 6.808 avaliações no mundo em 5.346 organizações, sendo 52% destas nos Estados Unidos e na China. Mais de 4.400 organizações foram certificadas entre os níveis 2 e 3 e mais de 370 organizações no nível 5 de maturidade. Mais de 4.100 organizações são do tipo comercial e *in-house*.<sup>72</sup>



**Figura 11.3** – Perfil das organizações com CMMI-DEV (versão 1.2) de 04/2002 a 06/2010.<sup>72</sup>

## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

Existem duas representações de CMMI-DEV (por estágios e contínua) que trazem resultados equivalentes. A escolha da representação a ser aplicada depende de algumas análises do ponto de vista da estratégia organizacional. Em primeiro lugar, deveria haver um plano de melhoria de qualidade. Baseado neste plano, a parte de desenvolvimento de software deverá ser focada. Comumente, inicia-se com um diagnóstico da situação atual (*as-is*) e planeja-se a situação futura (*to-be*). A situação futura pode ser atingida por metas intermediárias.

O CMMI-DEV por estágios proporciona melhor visibilidade da evolução dos processos da organização como um todo. Por exemplo, as organizações que estejam reconhecidamente no nível 1 de maturidade deveriam adotar esta representação do CMMI. O CMMI-DEV na representação contínua poderia ser adotado quando há um conjunto de processos eficazes

bem conhecidos e dominados e há clareza sobre quais processos seriam complementares aos existentes. Por exemplo, os processos de engenharia já são eficazes e os de gerenciamento de projeto são fracamente aplicados e há pouca visibilidade do andamento dos projetos. Neste caso, o foco seria nas áreas de processo de gerenciamento de projeto.

A implementação dos processos do CMMI-DEV passa por duas etapas importantes. A primeira é a elaboração dos processos e a segunda, a institucionalização dos processos na organização. A maturidade (CMMI-DEV por estágios) ou a capacidade (CMMI-DEV contínua) é atingida quando os processos fazem parte do dia a dia da organização.

Um projeto de melhoria de processos com o modelo CMMI-DEV (por estágios) dura em média de 18 a 24 meses para os níveis 2 e 3 individualmente e de 12 a 15 meses para os níveis 4 e 5 em conjunto. Os certificados de nível de maturidade são emitidos pelo SEI e têm validade de três anos.

## EXERCÍCIOS

1. O que significa uma organização estar em algum nível de maturidade?
2. O que significa um processo estar em algum nível de capacidade?
3. Se você tivesse que escolher entre as representações por estágios ou contínua do CMMI-DEV para um projeto de melhoria de processos, qual escolheria? Justifique.
4. Qual é a diferença entre áreas de processos (AP) e processos de software?
5. Uma empresa está realizando um projeto de melhoria de processos. No entanto, um desenvolvedor de software não realiza as atividades conforme um novo processo adotado e, sistematicamente, não usa os padrões definidos. Sem o comprometimento dele no processo de melhoria, os resultados não serão atingidos. Como deveria ser o convencimento deste desenvolvedor para aderir ao projeto de melhoria?
6. As empresas que se encontram no nível 1 (inicial) do modelo CMMI-DEV por estágios são caracterizadas entre outros por: ausência de processos de software, esforços individuais e gerentes que “apagam incêndio” frequentemente. Como as organizações poderiam mudar esta situação?

7. O modelo CMMI-DEV é um *framework* de boas práticas de Engenharia de Software visando à melhoria contínua de processos. Porém, as práticas podem ser vistas como muito burocráticas e difíceis de serem aplicadas no dia a dia. Como isso poderia ser minimizado?
8. Para realizar um projeto de melhoria de processos, o comprometimento dos níveis de gerência é fundamental. De que maneira este comprometimento poderia ser obtido?
9. As organizações que têm obtido certificações do modelo CMMI-DEV pertencem, na sua maioria, a grandes empresas. Em sua opinião, o modelo CMMI-DEV é compatível com a realidade de pequenas e médias empresas?
10. Algumas organizações procuram obter um certificado de nível de maturidade do CMMI-DEV pelo certificado em si ou por alguma exigência de mercado. Quais são as consequências desta iniciativa?

## SUGESTÕES DE LEITURA

HUMPHREY, Watts S. Characterizing the software process: a maturity framework. *IEEE Software*, vol.5, issue 2, pp. 73-79, Mar., 1988.

GILB, Thomas Level 6: why we can't get there from here. *IEEE Software*, vol. 13, issue 1, pp. 97-103, Jan., 1996.

GRISS, Martin L. CMM as a framework for adopting systematic reuse. *Object Magazine*, pp. 60-69, Mar., 1998.

PAULK, Mark C. Extreme programming from a CMM perspective. *IEEE Software*, vol. 18, issue 6, pp.19-26, Nov./Dec., 2001.

SAIEDIAN, Hossein; KUZARA, Richard. SEI capability maturity model's impact on subcontractors. *IEEE Computer*, vol. 28, issue1, pp. 16-26, Jan., 1995.

## 11.2. MR-MPS

*O Programa MPS-BR é uma iniciativa brasileira para melhoria contínua de processos de software. A Softex é responsável pela coordenação do Programa MPS-BR e é apoiada pelo Ministério da Ciência da Tecnologia (MCT), Financiadora de Estudos e Projetos (FINEP), Serviço Brasileiro de Apoio às Micro e Pequenas Empresas (SEBRAE) e Banco Interamericano de Desenvolvimento (BID). O MPS-BR surgiu para incentivar as empresas nacionais de pequeno e médio porte a terem acesso, a baixo custo, às boas práticas dos modelos de maturidade. O MPS-BR é constituído pelo Modelo de Referência (MR-MPS), Método de Avaliação (MA-MPS) e Modelo de Negócio (MN-MPS). O MR-MPS tem sido muito aceito ultimamente, existindo, inclusive, muitas empresas certificadas neste modelo.*

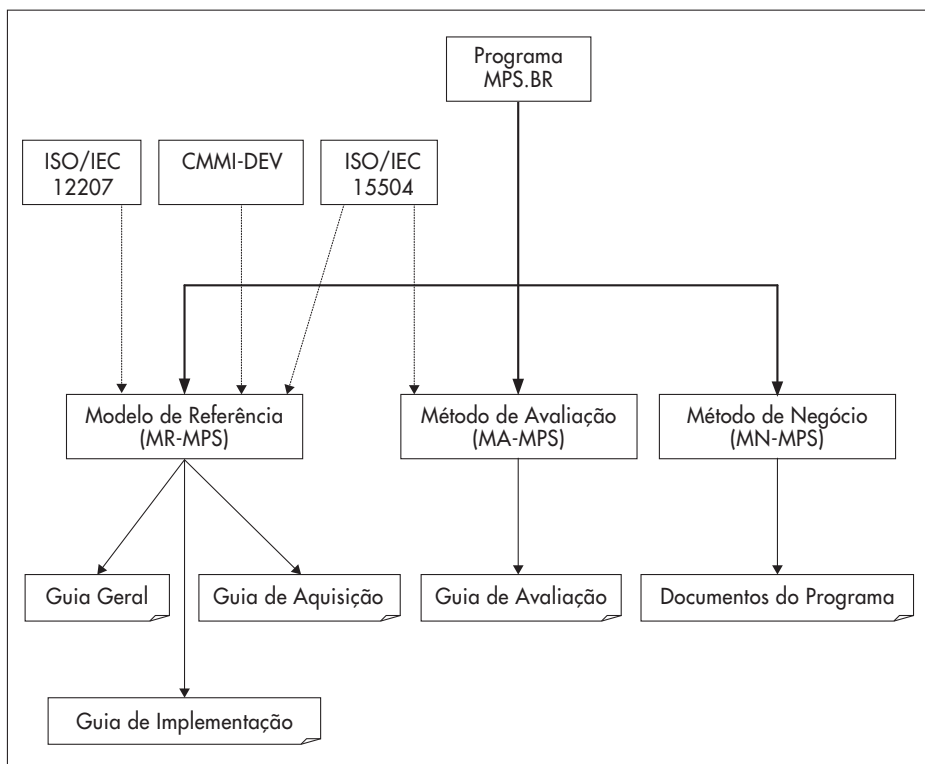
O Programa MPS-BR é uma iniciativa brasileira para melhoria contínua de processos de software criado em dezembro de 2003. O MPS-BR tem como uma de suas metas definir e aprimorar um modelo de melhoria e avaliação de processo de software, visando preferencialmente as micro, pequenas e médias empresas, de forma a atender suas necessidades de negócio e ser reconhecido nacional e internacionalmente como um modelo aplicável à indústria de software.<sup>51</sup>

A Softex – Associação para Promoção da Excelência do Software Brasileiro – é responsável pela coordenação do Programa MPS-BR e conta com o apoio do Ministério da Ciência e Tecnologia (MCT), Financiadora de Estudos e Projetos (FINEP), Serviço Brasileiro de Apoio às Micro e Pequenas Empresas (SEBRAE) e Banco Interamericano de Desenvolvimento (BID).<sup>51</sup>

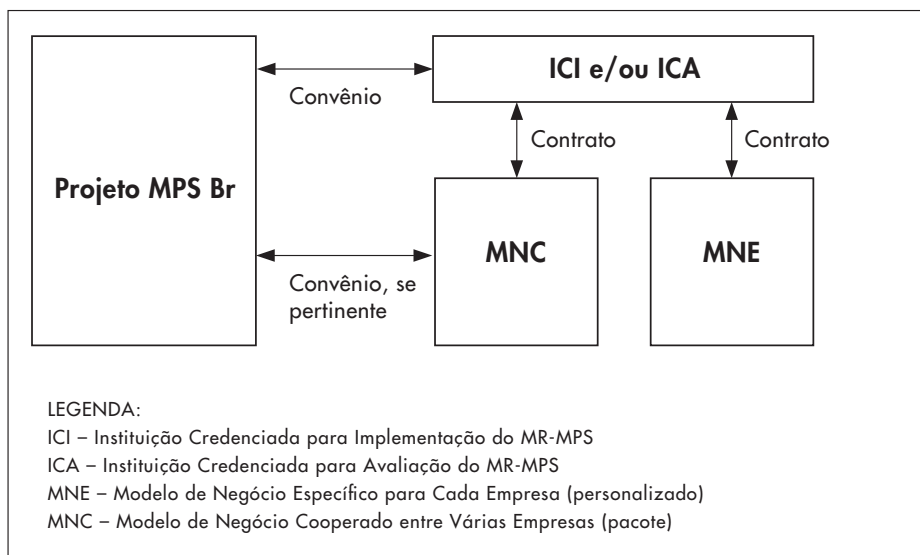
O MPS-BR é constituído dos seguintes componentes: Modelo de Referência (MR-MPS); Método de Avaliação (MA-MPS); e Modelo de Negócio (MN-MPS). O MR-MPS tem como base técnica as normas ISO/IEC 12207<sup>44, 45, 46</sup> e ISO/IEC 15504<sup>73</sup> e o modelo CMMI-DEV<sup>48, 49</sup>. O MA-MPS tem como base técnica a norma ISO/IEC 15504<sup>73</sup>. Cada um é constituído de guias e documentos que permitem a implementação/avaliação dos projetos de melhoria de processos. A Figura 11.4 apresenta o Programa MPS-BR e seus componentes.<sup>51</sup>

O Modelo de Negócio, ilustrado na Figura 11.5, foi definido para orientar as formas de participação das empresas no Programa MPS-BR. Existem duas formas de participação: Modelo de Negócio Específico (MNE) e Modelo de Negócio Cooperado (MNC).<sup>52</sup>

No MNE, uma empresa pode estabelecer um contrato diretamente com uma Instituição Credenciada para Implementação (ICI) e/ou uma Instituição Credenciada para Avaliação (ICA) para implementação e/ou avaliação, para ter o apoio ao seu projeto de melhoria de processo. Na forma cooperada, algumas empresas são reunidas em uma cooperativa que estabelecem um contrato com uma instituição credenciada que organiza esta participação e apoia a implementação/avaliação dos projetos de melhoria de processos da cooperativa.<sup>52</sup>

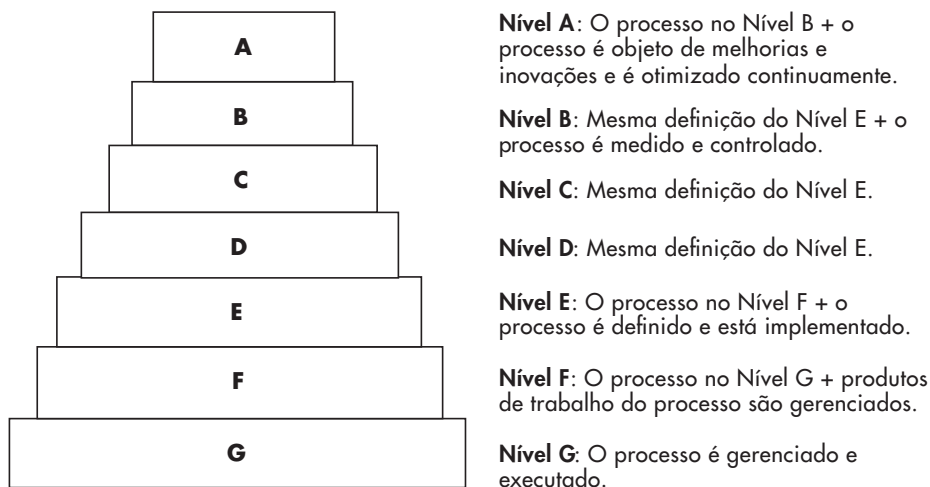


**Figura 11.4** – Componentes do Programa MPS-BR.<sup>51</sup>



**Figura 11.5 – Modelo de Negócio do MPS-BR.**<sup>52</sup>

O Modelo de Referência (MR-MPS) é constituído de sete níveis de maturidade, conforme apresentados na Figura 11.6.



**Figura 11.6 – Níveis de maturidade do MR-MPS.**<sup>51</sup>

A estrutura dos níveis de maturidade do MR-MPS é semelhante a do modelo CMMI-DEV por estágios, porém com sete níveis, sendo o nível G o nível inicial. Neste nível, os processos devem ser gerenciados e executados (atributos do nível). No nível F, além de atender aos atributos do nível G, os produtos de trabalho dos processos são gerenciados. No nível E, além de atender aos atributos do nível F, os processos são definidos e estão implementados. Nos níveis D e C, os processos devem atender aos mesmos atributos do nível E. No nível B, além de atender aos atributos do nível E, os processos são medidos e controlados. No nível A, além de atender aos atributos do nível B, os processos são objetos de melhorias e inovações e são otimizados continuamente.

A Tabela 11.2 apresenta os processos do MR-MPS relacionados a cada um dos nível de maturidade.

**Tabela 11.2** – Processos do MR-MPS relacionados a cada um dos níveis de maturidade.<sup>51</sup>

NÍVEIS	PROCESSOS
A – Em otimização	Não há processos definidos.
B – Quantitativamente gerenciado	Gerência de Projetos – GPR (evolução)
C – Definido	Gerência de Decisões – GDE Gerência de Riscos – GRI Desenvolvimento para Reutilização – DRU
D – Largamente definido	Desenvolvimento de Requisitos – DRE Projeto e Construção do Produto – PCP Integração do Produto – ITP Verificação – VER Validação – VAL
E – Parcialmente definido	Avaliação e Melhoria do Processo Org. – AMP Definição do Processo Organizacional – DFP Gerência de Recursos Humanos – GRH Gerência de Reutilização – GRU Gerência de Projeto – GPR (evolução)
F – Gerenciado	Medição – MED Gerência de Configuração – GCO Aquisição – AQU Garantia da Qualidade – GQA Gerência de Portfólio de Projetos – GPP
G – Parcialmente gerenciado	Gerência de Requisitos – GRE Gerência de Projetos – GPR



## REFLEXÃO SOBRE OS CONCEITOS APRESENTADOS

O Programa MPS-BR é uma iniciativa brasileira de melhoria de processos de software. O seu modelo de referência (MR-MPS) tem uma estrutura semelhante a do modelo CMMI-DEV por estágios e é mais abrangente e detalhada. Porém, a estrutura do MR-MPS tem mais níveis que facilitam a implementação gradual dos processos em prazos menores.

No entanto, exige por parte da organização um comprometimento real sobre os resultados a serem atingidos com o MR-MPS. Do mesmo modo que o modelo CMMI-DEV, a implementação dos processos passa por dois passos: a elaboração e a institucionalização dos processos na organização.

Um dos atrativos para a implementação do modelo MR-MPS é a participação do governo brasileiro com a Softex através de financiamento de parte dos custos de implementação.

As iniciativas de melhorias de processos (baseadas em normas e modelos) tem um obstáculo comum que é cultura organizacional. Ela precisa ser administrada através da perspectiva de gerenciamento de mudanças organizacionais para ter sucesso nas certificações de qualidade.

Os certificados de nível de maturidade são emitidos pela Softex e têm validade de três anos.

## EXERCÍCIOS

1. O governo brasileiro tem incentivado as organizações brasileiras a implementarem o MR-MPS. Quais são as vantagens de implementar um modelo brasileiro?
2. Se você fosse escolher entre os modelos CMMI-DEV e MR-MPS, qual escolheria? Justifique.
3. Os modelos de maturidade CMMI-DEV e MR-MPS são similares. Qual é a relação entre os seus níveis de maturidade?
4. Por que o MR-MPS é mais adequado para pequenas e médias empresas?
5. Pesquise os processos do nível G e F do MR-MPS. Quais são as semelhanças e diferenças em relação aos processos do nível 2 do modelo CMMI-DEV? (Dica: pesquise no endereço [www.softex.br/mpsbr](http://www.softex.br/mpsbr).)

6. Muitas organizações de software estão implementando métodos ágeis nos seus processos de desenvolvimento. O MR-MPS tem muitas práticas a serem atendidas. Em sua opinião, as duas abordagens são compatíveis?
7. Sugira uma forma de gerenciar as mudanças organizacionais, em função de um projeto de melhoria de processos.
8. Por que no nível A não há processos definidos no modelo MR-MPS?
9. O GPP (nível F) e o GPR (nível G) estão relacionados. De que forma estão relacionados? (Dica: pesquise no endereço [www.softex.br/mpsbr](http://www.softex.br/mpsbr).)
10. Qual é a importância da validade de três anos de um certificado de nível de maturidade no modelo MR-MPS?

## SUGESTÕES DE LEITURA

FERNANDES, Patrícia G.; OLIVEIRA, Juliano L.; MENDES, Fabiana F.; SOUZA, Adriana S. *Resultados de implementação cooperada do MPS.BR*. In: III Workshop de Implementadores (W2 – MPS.BR), Nov., 2007, Belo Horizonte.

PRIKLANDINICKI, Rafael; MAGALHÃES, Ana Liddy C. C. *Implantação de modelos de maturidade com metodologias ágeis: um relato de experiências*. In: VI Workshop Anual do MPS (WAMPS), Out., 2010, Campinas, pp. 88-98.

SANTOS, Gleison; MONTONI, Mariano; VASCONCELLOS, Jucele; FIGUEIREDO, Sávio; CABRAL, Reinaldo; CERDEIRAL, Cristina; KATSURAYAMA, Anne Elise; NATALI, Ana Candida; LUPO, Peter; ZANETTI, David; ROCHA, Ana Regina. *Implementação do MR-MPS Níveis G e F em grupos de empresas do Rio de Janeiro*. In: III Workshop de Implementadores (W2 – MPS.BR), Nov., 2007, Belo Horizonte.

SANTOS, Gleison.; KIVALWEBER, Kival C. W.; ROCHA, Ana. Regina. C. Software process improvement in Brazil: evolving the MPS model and consolidating the MPS.BR program. In: XXXV Conferência Latinoamericana de Informática (CLEI), Set., 2009, Pelotas, pp. 1-11.

THIRY, Marcelo; WANGENHEIM, Christiane. G; ZOUCAS, Alessandra. *Implementação do MPS.BR em grupo de empresas da ACATE em Florianópolis 2007/2008*. In: III Workshop de Implementadores (W2 – MPS.BR), Nov., 2007, Belo Horizonte.

WEBER, Kival C.; ARAÚJO, Eratóstenes; MACHADO, Cristina A. F.; SCALET, D.; SALVIANO, Clênio F.; ROCHA, Ana Regina C. Modelo de referência e método de

avaliação para melhoria de processo de software – versão 1.0 (MR-MPS e MA-MPS). In: *IV Simpósio Brasileiro de Qualidade de Software*, Jul., 2005, Porto Alegre.

YOSHIDA, David; KOHAN, Sarah; SALVETTI, Nilson; HIRAMA, Kechi. Convênio Com-Qualidade: implementação MPS.BR nível G em grupo de empresas de São Paulo. In: *III Workshop de Implementadores (W2 – MPS.BR)*, Nov. 2007, Belo Horizonte.

## Considerações finais

---

*Os conceitos e as técnicas discutidos neste livro são fundamentais no desenvolvimento de software. Investimentos adequados em processos, pessoas e tecnologias trazem benefícios para organização, melhorando a maturidade dos seus processos e proporcionando qualidade e produtividade de software. Como pode ser visto no livro, a disciplina de Engenharia de Software continuará a evoluir para apoiar as demandas cada vez mais exigentes dos negócios.*

Nos dias atuais, em que as necessidades de negócio são cada vez mais exigentes, desenvolver sistemas de software passou a ser um desafio constante. Historicamente, observa-se uma evolução das abordagens e tecnologias que foram desenvolvidas para dar melhor apoio ao desenvolvimento de software que atendam não somente ao negócio, mas também às restrições de custo, prazo e qualidade.

Assim, o livro apresentou inicialmente a disciplina de Engenharia de Software, que fornece o apoio conceitual, técnico e metodológico para tratar o desenvolvimento de software como um assunto de engenharia. Dentre estes apoios, foram destacados os métodos de desenvolvimento de software, seguindo as abordagens Estruturada e Orientada a Objetos.

Também foram tratados os processos para orientar as atividades que devem ser realizadas para obter um software com qualidade. Assim, foram destacadas as atividades de **gerenciamento** e **desenvolvimento** de

software que são frequentemente encontradas em vários tipos de processos de software.

Outro tema apresentado foi o desenvolvimento de **interface com o usuário**, algo que é muito importante nos dias atuais, em função de muitos sistemas de software possuírem a característica de interatividade (vide aplicações bancárias, internet etc.). Neste contexto, o usuário passou a desempenhar um papel fundamental nas funções processadas pelo software.

Todos os métodos e processos de gerenciamento e de desenvolvimento de software não bastam se processos de **verificação, validação, configuração e qualidade** não estiverem presentes.

Para obter software com qualidade, são necessários processos, pessoas e tecnologias (métodos, técnicas e ferramentas) para apoiar as atividades de desenvolvimento. Assim, neste cenário, as ferramentas CASE foram discutidas. Diversas aplicações de software são complexas, portanto, para ter um melhor controle no desenvolvimento de software e obter produtividade é imprescindível o uso de ferramentas.

Atualmente, estão disponíveis outras **tecnologias** de desenvolvimento de software que fazem frente aos vários tipos de aplicações que surgem a cada dia. Entre essas tecnologias, foram discutidas também o reuso de software e a arquitetura orientada a serviços. Muitas tecnologias estão sendo pesquisadas nas universidades e na iniciativa privada como a Orientação a Aspectos (*Aspect-Oriented Development*) para tratar melhor as necessidades de negócio relacionadas com requisitos não funcionais.

Uma organização de software precisa estruturar as suas áreas de gerenciamento, desenvolvimento e apoio para, em conjunto, entregar o software com qualidade e dentro de restrições de prazo e custo acordados. Além disso, a organização precisa que o mercado a reconheça pela aplicação de boas práticas no desenvolvimento de software. Os modelos de maturidade são usados como referências para essas boas práticas. Assim, foram destacados os modelos CMMI-DEV e MR-MPS. Uma organização pode obter certificações nesses modelos através de entidades credenciadas.

Em resumo, os conceitos e as técnicas discutidos neste livro são fundamentais no desenvolvimento de software. Investimentos adequados em processos, pessoas e tecnologias trazem benefícios para organização, melhorando a maturidade dos seus processos e promovendo a qualidade e a produtividade do software.



## GLOSSÁRIO

---

### **Arquitetura**

Uma arquitetura apresenta uma visão dos elementos estruturais e comportamentais do sistema.

### **Aspecto**

É um novo tipo de abstração usado no desenvolvimento de software que implementa um interesse transversal.

### **Caso de Uso**

É uma técnica para especificar uma função do software do ponto de vista de um ator (usuário ou outro sistema).

### **Ciclo de Vida**

O período de tempo que inicia quando um produto de software é concebido e termina quando o software não é mais disponível para o uso. O ciclo de vida de software tipicamente inclui uma fase de concepção, requisitos, projeto, implementação, teste, instalação e verificação, operação e manutenção, e algumas vezes, retirada de operação.

### **Etnografia**

É uma técnica de observação do usuário nas atividades de seu dia a dia no trabalho, observando suas interações com os outros e o uso de recursos do ambiente. Pode ser usada para elicitación e análise de requisitos.

**Ferramenta**

CASE (*Computer Aided Software Engineering*) é um programa de computador usado no desenvolvimento, teste, análise ou manutenção de um programa ou sua documentação.

**JAD (*Joint Application Development*)**

É uma técnica antiga usada para coletar informações dos clientes por meio de reuniões para juntos chegarem a um entendimento do problema. Protótipos e diagramas podem ser usados para este entendimento.

**Método**

Um padrão que descreve ordenadamente as características de um processo ou procedimento usado na engenharia de um produto ou realização de um serviço.

**Método Ágil**

Tem como características a aceitação de mudanças nos seus requisitos, foco na colaboração entre clientes e desenvolvedores e apoio na entrega antecipada do produto. Exemplos de métodos ágeis são XP (*Extreme Programming*) e Scrum.

**Modelo de Maturidade**

É uma representação de um conjunto de processos organizacionais que contém boas práticas e metas com o propósito de torná-los mais previsíveis e aumentar a qualidade dos seus produtos.

**Modelo de Processo**

É uma representação que reúne processos, métodos e ferramentas, e fases genéricas. É uma abstração do processo real que está sendo executado.

**Processo**

É um conjunto (do inglês *framework*) de atividades que são aplicadas a todos os projetos de software, sem considerar o seu tamanho ou complexidade. É uma série de estágios que envolvem atividades, controles e recursos para produzir uma saída de alguma espécie.



**Processo de Software**

É um conjunto de atividades voltadas para o desenvolvimento e evolução de um software. O processo envolve a tradução das necessidades do usuário em requisitos de software, transformação de requisitos em projeto, implementação do projeto em código, teste de código e, algumas vezes, instalação e verificação de software para uso operacional. Por exemplo, desenvolvimento incremental, prototipação rápida, modelo espiral, modelo cascata.

**Produto de Software**

Um conjunto completo de programas de computador, procedimentos e possivelmente documentação e dados associados projetados a serem liberados para um usuário.

**Prova de Conceito (PoC – *Proof of Concept*)**

É uma técnica para verificar se uma ideia ou teoria pode dar resultados esperados. Em desenvolvimento de software, pode-se verificar se o software é exequível. É comum desenvolver protótipos funcionais com esse propósito.

**Release**

Versão do software gerado para um propósito específico que será distribuído aos clientes.

**Requisito**

Uma declaração de algo que precisa ser implementado no sistema. Um requisito funcional é uma característica ou função do software. Um requisito não funcional é uma restrição ou comportamento esperado que se aplica a todo o sistema.

**Reuso**

É uma tecnologia para aumentar a produtividade com o reuso de sistemas ou componentes existentes.

**Separação de Interesses**

Estabelece que o software deva ser organizado de tal forma que cada elemento de programa faça uma e somente uma coisa.

**Serviço**

No âmbito de uma arquitetura SOA, é um elemento computacional que tem como propósito desempenhar uma função específica e que pode ser utilizado por um cliente.

**SOA (*Service Oriented Architecture*)**

É uma tecnologia que torna possível, além do reuso de aplicações, o acesso a informações diversas disponíveis em vários computadores ligados na internet.

**Stakeholder**

Pessoa ou organização que está ativamente envolvida no projeto cujos interesses podem ser afetados durante a execução do projeto.

**Técnica**

Descreve as características acumuladas na aplicação de habilidades e métodos técnicos e gerenciais na criação de um produto ou realização de um serviço.

**UML (*Unified Modeling Language*)**

É uma linguagem visual resultante de numerosos métodos orientados a objetos que existiam no início da década de 1990. Fornece diferentes visões de um sistema por meio de seus diversos tipos de modelos.

**Validação**

Processo que permite avaliar se um sistema atende às necessidades e expectativas do cliente.

**Verificação**

Processo que permite avaliar se um sistema atende à sua especificação.

## NOTAS BIBLIOGRÁFICAS

---

1. NAUR, Peter; RANDELL, Brian. (Eds.). *Software engineering: a report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968*. Brussels: Scientific Affairs Division, NATO, 1969.
2. IEEE. *ANSI/IEEE Std 610.12-1990 – IEEE standard glossary of software engineering terminology*. IEEE Standards Collection. Software Engineering. New York: IEEE, 1990.
3. PRESSMAN, R. S. *Software Engineering – A Practitioner's Approach*. 6<sup>th</sup>. edition. McGraw Hill. 2007. 880p.
4. CARVALHO, Marli M.; PALADINI, Edson. P. et al. *Gestão da qualidade – teoria e casos*. Rio de Janeiro: Campus, 2006.
5. ARTHUR, Lowell. J. *Melhorando a qualidade do software – Um guia completo para o TQM*. Rio de Janeiro: IBPI Press, 1994.
6. EVELEENS, J. Laurenz; VERHOEF, Chris. The rise and fall of the chaos report figures. *IEEE Software*, vol. 27, issue 1, pp. 30-36, Sept., 2010.
7. BOURQUE, Pierre; PUIS, Robert; TRIPP, Leonard L. (Eds.). *Guide to the Software Engineering Body of Knowledge – SWEBOK*. USA: IEEE Computer Society Press, 2004.
8. SOMMERVILLE, Ian. *Engenharia de software*. Tradução: Kalinka Oliveira e Ivan Bosnic. Revisão Técnica: Kechi Hirama. 9. ed. São Paulo: Prentice Hall, 2011.
9. BOEHM, Barry. W. A spiral model of software development and enhancement. *IEEE Computer*, vol. 21, issue 5, pp. 61-72, May. 1988.
10. JACOBSON, Ivar; BOOCH, Grady; RUMBAUGH, James. *The unified software development process*. USA: Addison-Wesley Professional, 1999.
11. BECK, Kent et al. *The agile manifesto*. 2001. Disponível em: <<http://agilealliance.org>>. Acesso em: 10 jun. 2011.

12. BECK, Kent; CYNTHIA, Andres. *Extreme programming explained: embrace change*. 2<sup>nd</sup> edition. USA: Addison-Wesley Professional, 2004.
13. SCHWABER, Ken. *Agile project management with Scrum*. USA: Microsoft Press, 2004.
14. WELLS, James. D. *Extreme programming: a gentle introduction*. 1999. Disponível em: <<http://www.extremeprogramming.org>>. Acesso em: 15 jun. 2011.
15. PMI. *Um guia do conjunto de conhecimentos em gerenciamento de projetos. PMBOK* 4. ed. São Paulo: Project Management Institute, 2008.
16. KOTONYA, Gerald; SOMMERVILLE, Ian. *Requirements engineering: process and techniques*. Chichester: Wiley, 1998.
17. SOFTEX. *Guia de implementação – parte 1: fundamentação para implementação do nível G do MR-MPS*. SOFTEX, Julho, 2011.
18. DeMARCO, Tom. *Análise estruturada e especificação de sistemas*. Rio de Janeiro: Campus, 1989.
19. GANE, Chris; SARSON, Trish. *Análise estruturada de sistemas*. Rio de Janeiro: LTC, 1983.
20. YOURDON, Edward. *Análise estruturada moderna*. Rio de Janeiro: Campus, 1990.
21. RUMBAUGH, James; BLAHA, Michael; PREMERLANI, William; EDDY, Frederick; LORENSSEN, William. *Object-oriented modeling and design*. Englewood Cliffs: Prentice Hall, 1991.
22. JACOBSON, Ivar.; CHRISTERSON, Magnus; JONSSON, Patrik; ÖVERGAARD, Gunnar. *Object-oriented software engineering: a use case driven approach*. USA: Addison Wesley Professional, 1992.
23. BOOCH, Grady. *Object-oriented analysis and design with applications*. 2<sup>nd</sup>. Edition. USA: Addison-Wesley Professional, 1994.
24. BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. *The unified modeling language user guide*. 2<sup>nd</sup> edition. USA: Addison-Wesley Professional, 2005.
25. FOWLER, Martin. *UML essencial*. 3. ed. São Paulo: Bookman, 2005.
26. PRESSMAN, Roger S. *Software engineering – a practitioner's approach*. 6<sup>th</sup> edition. New York: McGraw Hill, 2007.
27. IEEE. *IEEE Std. 830-1993 – Recommended practice for software requirements specifications*. IEEE Computer Society, November, 1993.
28. BASS, Len; CLEMENTS, Paul; KAZMAN, Rick. *Software architecture in practice*. 2<sup>nd</sup> edition. USA: Addison-Wesley Professional, 2003.

29. GARLAN, David; SHAW, Mary. *An introduction to software architecture*. CMU-CS-94-166. Carnegie Mellon University, 1994.
30. KRUCHTEN, Philippe. Architectural blueprints – the “4+1” view model of software architecture. *IEEE Software*, vol. 12, no. 6, pp. 42-50, November, 1995.
31. KERNIGHAN, Brian. W.; PLAUGER, P. J. *The elements of programming style*. 2nd edition. USA: McGraw-Hill, 1978.
32. HETZEL, Bill. *The complete guide to software testing*. 2nd edition. Chichester: Wiley, 1993.
33. MYERS, Glenford J. *The art of software testing*. 2<sup>nd</sup> edition. USA: Wiley, 2004.
34. DIJKSTRA, Edsger. W. *Notes on structured programming*. Technological University Eindhoven, The Netherlands, Department of Mathematics, 1969.
35. IEEE. *ANSI/IEEE Std 829-2008 – IEEE standard for software and system test documentation*. IEEE, July, 2008.
36. SOMMERVILLE, Ian. *Engenharia de software*. Tradução: Selma Shin Shimizu Melnikoff, Reginaldo Arakaki, Edilson de Andrade Barbosa. Revisão Técnica: Kechi Hiramã. 8. ed. São Paulo: Prentice Hall, 2011.
37. ABNT. *NBR 9241-11. Requisitos ergonômicos para trabalho de escritórios com computadores – Parte 11: orientações sobre usabilidade*. ABNT, 2002.
38. HUMPHREY, Watts S. *Managing the software process*. USA: Addison-Wesley Professional, 1989.
39. PERRY, William E. *Effective methods for software testing*. New York: Wiley, 1995.
40. DELAMARO, Márcio. E; MALDONADO, José C.; JINO, Mario. *Introdução ao teste de software*. Rio de Janeiro: Campus, 2007.
41. FAGAN, Michael E. Advances in software inspection. *IEEE Transactions on Software Engineering*, vol. 12, issue 7, pp. 744-751, Jul., 1986.
42. WHEELER, David A.; BRYKCZYNSKI, Bill; MEESON Jr., Reginald. N. *Software inspection – an industry best practice*. USA: IEEE Computer Society Press, 1996.
43. FAGAN, Michael E. Design and code inspections to reduce errors in program development. *IBM System Journal*, vol. 15, issue 3, pp. 182-211, 1976.
44. ISO. *ISO/IEC 12207 Information technology – Software life cycle processes*. ISO. 1995.
45. ISO. *ISO/IEC 12207 Information technology –software life cycle processes – amendment 1*. ISO. 2002.

46. ISO. ISO. *ISO/IEC 12207 Information technology –software life cycle processes – amendment 2*. ISO. 2004.
47. IEEE. IEEE. *ANSI/IEEE Std 828-1990 – IEEE standard for software configuration management plans*. New York: IEEE, 1990.
48. IEEE. *ANSI/IEEE 1042-1987 – IEEE guide to software configuration management*. New York: IEEE, 1987.
49. CHRISSIS, Mary B.; KONRAD, Mike; SHRUM, Sandra. *CMMI for development – guidelines for process integration and product improvement*. 3rd. edition. USA: Addison-Wesley Professional, 2011.
50. SEI. *CMMI for Development (CMMI-DEV)*. Version 1.3. CMU/SEI-2010-TR-033, Nov., 2010.
51. SOFTEX. *MPS.BR – Guia geral*. Versão 2011. SOFTEX, Jun., 2011.
52. WEBER, Kival C.; ARAÚJO, Eratóstenes; MACHADO, Cristina A. F.; SCALET, D.; SALVIANO, Clênio F.; ROCHA, Ana Regina C. *Modelo de referência e método de avaliação para melhoria de processo de software – versão 1.0 (MR-MPS e MA-MPS)*. In: IV Simpósio Brasileiro de Qualidade de Software, Jul., 2005, Porto Alegre.
53. ISO. *Quality management system – requirements*. ISO 9001. 2008.
54. ISO. *Software and system engineering – guidelines for the application of ISO 9001:2000 to computer software*. ISO/IEC 90003:2004, 2004.
55. IEEE. *IEEE Std 730-2002 – IEEE Standard for software quality assurance plans*. IEEE, September, 2002.
56. PRIETO-DIAZ, Ruben. *Reuse as a new paradigm for software development. systematic reuse: issues in initiating and improving a reuse program*. In: Proceedings of the International Workshop on Systematic Reuse. London, Jan., 1996.
57. FRANKS, William B.; ISODA, Sadahiro. Success factors of systematic reuse. *IEEE Software*, vol. 11. issue 5, pp. 14-19, Sept., 1994.
58. LUCRÉDIO, Daniel; BRITO, Kellyton S.; ALVARO, Alexandre; GARCIA, Vinicius C.; ALMEIDA, Eduardo S.; FORTES, Renata P. M.; MEIRA, Silvio L. Software reuse: the brazilian industry scenario. *The Journal of Systems and Software*, vol. 81, issue 6, pp. 996-1013, Jun., 2008.
59. FUGITA, Henrique S. *MAPOS: método de análise e projeto orientado a serviços*. Escola Politécnica da USP, 2009, 175p. Dissertação de Mestrado em Sistemas Digitais, São Paulo, 2009.

60. HUHNS, Michael. N.; SINGH, Munindar P. *Service-oriented computing: key concepts and principles*. *IEEE Internet Computing*, vol. 9. issue. 1, pp. 75-81, Jan./Feb., 2005.
61. NUSEIBEH, Bashar. *Crosscutting requirements*. In: AOSD'04 International Conference on Aspect-Oriented Software Development. Lancaster, UK, Mar., 2004. pp. 3-4.
62. DIJKSTRA, Edsger. W. *On the role of scientific thought*. Selected Writings on Computing: A Personal Perspective. Springer-Verlag, 1982, p.60-66.
63. PARNAS, David L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, vol. 15, issue 12, pp. 1053-1058, Dec., 1972.
64. GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1994.
65. KICZALES, Gregor; LAMPING, John; MENDHEKAR, Anurag; MAEDA, Chris; LOPES, Cristina V.; LOINGTIER, Jean-Marc. M.; IRWIN, John. *Aspect-oriented programming*. In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag LNCS 1241, Jun., 1997.
66. KATZ, Shmuel. *A survey of verification and static analysis for aspects*. *AOSD-Europe-Technion-1*, M8.1, Jul., 2005.
67. KICZALES, Gregor; HILSDALE, Erik.; HUGUNIN, Jim.; KERSTEN, Mik; PALM, Jeffrey; GRISWOLD, William G. *An overview of AspectJ*. In: *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 2001.
68. KICZALES, Gregor; HILSDALE, Erik; HUGUNIN, Jim; KERSTEN, Mik; PALM, Jeffrey; GRISWOLD, William G. Getting started with AspectJ. *Communications of the ACM*. vol. 44, issue 10, pp. 59-65, Oct., 2001.
69. ECLIPSE. *AspectJ*. Disponível em: <<http://www.eclipse.org/aspectj/index.php>>. Acesso em: 20 jul. 2011.
70. SEI. *CMMI for Acquisition (CMMI-ACQ)*. Version 1.3. CMU/SEI-2010-TR-032, Nov., 2010.
- 71 SEI. *CMMI for service (CMMI-SRV)*. Version 1.3. CMU/SEI-2010-TR-034, Nov., 2010.
72. *SEI CMMI for development SCAMPI Class A. Appraisal results 2010 end-year update*. Software Engineering Institute (SEI) – Carnegie Mellon University, Mar., 2011.
73. ISO. *ISO/IEC 15504-2: Information Technology – Process Assessment – Part 2 – Performing an Assessment*. ISO. 2003.

## BIBLIOGRAFIA COMPLEMENTAR

---

ACKERMAN, A. F.; BUCHWALD, L. S.; LEWSKI, F. H. Software inspections: an effective verification process. *IEEE Software*, vol. 6, issue 3, pp. 31-36, 1989.

ADRION, W. R.; BRANSTED, M. A.; CHERNIAVSKY, J. C. Validation, verification and testing of computer software. *Computing Surveys*, vol. 14, issue 2, Jun., 1982.

BARTEL, T.; FINSTER, M. A TQM Process for systems integration. *Information System Management*. Summer, pp. 19-29, 1995.

BECK, K. *Embracing change with extreme programming*. IEEE Computer, vol. 32, issue 10, pp. 70-77, Oct., 1999.

BERKUN, S. *A Arte do gerenciamento de projetos*. Tradução: Carlos A. C. Moraes, Tereza C. F. Souza. Revisão Técnica: Henrique J. Brodbeck. O'Reilly: Bookman, 2008.

BERSOFF, E. H.; DAVIS, A M. Impacts of life cycle models on software configuration management. *Communications of the ACM*. vol. 34, issue 8, pp.105-18, Aug., 1991.

BERSOFF, E. H.; HENDERSON, V. D.; SIEGEL, S. G. Software configuration management: a tutorial. *Computer*, vol. 12, issue 1, pp. 6-14, Jan., 1979.

BINDER, R. V.; *Design for testability in object-oriented systems*. *Communications of the ACM*, vol.87, issue 9, pp. 87-101, Sep., 1994.

BOEHM, B. W. Anchoring the software process. *IEEE Software*, pp.73-82.Jul., 1996.

BOEHM, B. W. Get ready for agile methods, with care. *IEEE Computer*, vol. 35, issue 1, pp. 64-69, 2002.

BOEHM, B. W. Improving software productivity. *IEEE Computer*, vol. 20, issue 9, pp. 43-57, 1987.



- BOEHM, B. W. et al. A software development environment for improving productivity. *IEEE Computer*, vol.17, issue 6, pp. 30-42, 1984.
- CARD, D.; COMER, E. Why do so many reuse programs fail? *IEEE Software*. vol. 11, issue 5, pp. 114-115, Sep., 1994.
- COCKBURN, A. Selecting a project's methodology. *IEEE Software*, vol.17, issue 4, pp. 64-71, 2000.
- DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on test data selection: help for the practicing programmer. *IEEE Computer*, vol.11, issue 4, pp. 34-41, Apr., 1978.
- ERL, T. *Service-oriented architecture: concepts, technology and design*. USA: Prentice Hall, 2005.
- ERL, T. *SOA Principles of service design*. USA: Prentice Hall, 2007.
- FELDMAN, S. I. Software configuration management – past uses and future challenges. *Lecture notes in Computer Science*, vol. 550, pp.1-6, 1991.
- FERNANDES, P. G.; OLIVEIRA, J. L.; MENDES, F. F.; SOUZA, A. S. Resultados de implementação cooperada do MPS.BR. III Workshop de Implementadores (W2 – MPS.BR). Belo Horizonte (MG), nov., 2007.
- FISCHER, A. S. *CASE: Using software development tools*. 2<sup>nd</sup> Edition. Wiley, 1991.
- FRAKES, W. B.; FOX, C. J. Sixteen questions about software reuse. *Communications of ACM*, vol. 38, issue. 6, pp. 75-87, Jun., 1995.
- FRAKES, W.; TERRY, C. Software reuse: metrics and models. *ACM Computing Surveys*, vol. 28, issue 2, pp. 415-35, Jun., 1996.
- FRANKL, P. G.; WEYUKER, E. J. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, vol. 14, issue 10, pp. 1483-1498, Oct., 1988.
- GILB, T. Level 6: Why we can't get there from here. *IEEE Software*, vol. 13, issue 1, pp. 97-103, 1996.
- GOODENOUGH, J. B.; GERHART, S. L. Toward a theory of test data selection. In: *International Conference on Reliable Software*, vol.10, issue 6, pp. 493-510, Jun., 1975.
- GRISS, M. L. CMM as a framework for adopting systematic reuse. *Object Magazine*, pp.60-69, Mar. 1998.
- HIGHSMITH, J.; COCKBURN, A. Agile software development: the business of innovation. *IEEE Computer*, vol. 34, issue 9, pp. 120-122, 2001.
- HONG, Z.; HALL, P. A. V.; MAY, J. H. R. Software unit test coverage and adequacy. *ACM Computer Survey*, vol. 29, issue 4, pp. 365-427, Dec., 1997.

- HUMPHREY, W. S. Characterizing the software process: a maturity framework. *IEEE Software*, vol. 5, issue 2, pp. 73-79, Mar., 1988.
- IEEE. Std 1028-1988 – *IEEE Standard for software reviews and audits*. 1988.
- ISO. ISO/IEC 12207 *System and software engineering – software life cycle processes*. ISO. 2008. Disponível em: <<http://www.iso.org>>. Acesso em: 5 Set. 2011.
- JACOBSON, I.; NG, P.-W. *Aspect-oriented software development with use cases*. USA: Addison-Wesley – Pearson Education, 2005.
- KERNIGHAN, B. W.; PIKE, R. *The practice of programming*. USA: Addison-Wesley, 1999.
- KENETT, R. S.; KOENIG, S. A process management approach to SQA. *Quality Progress*, pp.66-70, Nov., 1988.
- LEE, G. *Object-oriented GUI application development*. New Jersey: Prentice Hall, 1993.
- LIM, W. C. Effects of reuse on quality, productivity and economics. *IEEE Software*, pp. 23-30, Sep., 1994.
- McMENAMIM, S. M.; PALMER, J. F. *Análise essencial de sistemas*. USA: McGraw-Hill, 1991.
- MEYER, B. Reusability: The case for object-oriented design. *IEEE Software*, vol. 4, issue 2, pp. 50-64., Mar., 1987.
- MISFELDT, T.; BUMGARDNER, G.; GRAY, A.; XIAOPING, L. *The elements of C++ Style*. USA: Cambridge University Press, 2004.
- NTAFOS, S. C. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, vol. 14, issue 6, pp. 868-874, Jun. 1988.
- PARNAS, D. L.; LAWFORD, M. The role of inspection in software quality assurance. *IEEE Transactions on software engineering*. vol. 29, issue 8, pp. 674-676. Aug., 2003.
- PAULK, M. C. Extreme programming from a CMM perspective. *IEEE Software*, vol. 18, issue 6, pp. 19-26, Nov./Dec., 2001.
- PORTER, A., VOTTA, L. What makes inspections work? *IEEE Software*, vol. 14, issue 6, pp. 99-102, Nov., 1997.
- PRIKLANDINICKI, R.; MAGALHÃES, A. L. Implantação de modelos de maturidade com metodologias ágeis: um relato de experiências. In: *VI Workshop Anual do MPS (WAMPS)*. Campinas – SP, Brasil. Out. 2010. pp. 88-98.
- REIFER, D. J. How good are agile methods?. *IEEE Computer*, vol. 19, issue 4, pp. 16-18, Aug., 2002.
- ROYCE, W. *Software project management – a unified framework*. USA: Addison-Wesley, 1998.

ROYCE, W. W. Managing the development of large software systems. In: *Proceedings IEEE WESCON*, pp. 1-9, Aug., 1970.

ROYER, T. C. *Software testing management: life on the critical path*. USA: Prentice Hall, 1993.

RUBIN, J.; CHISNELL, D. *Handbook of usability testing: how to plan, design, and conduct effective tests*. 2<sup>nd</sup> Edition. Wiley, 2008.

SAIEDIAN, H.; KUZARA, R. SEI capability maturity model's impact on subcontractors. *IEEE Computer*, vol. 28, issue 1, pp. 16-26, 1995.

SANTOS, G.; MONTONI, M.; VASCONCELLOS, J.; FIGUEIREDO, S.; CABRAL, R. C.; CERDEIRAL, C.; KATSURAYAMA, A. E.; NATALI, A. C.; LUPO, P.; ZANETTI, D.; ROCHA, A. R. Implementação do MR-MPS níveis G e F em grupos de empresas do Rio de Janeiro. *III Workshop de Implementadores (W2 – MPS.BR)*. Belo Horizonte (MG), Nov., 2007.

SANTOS, G.; KIVAL, C. W.; ROCHA, A. R. C. Software process improvement in Brazil: evolving the mps model and consolidating the MPS.BR Program. In: XXXV Conferência Latino-americana de Informática (CLEI). Pelotas-RS, Brasil, Set., 2009. pp.1-11.

SCHULMEYER, G. G.; McMANUS, J. I. *Handbook of software quality assurance*. 3<sup>rd</sup> edition. USA: Prentice Hall, 1999.

SHARON, D.; ANDERSON, T. A Complete software engineering environment. *IEEE Software*, vol. 14, issue 2, Mar./Apr., 1997.

THIRY, M.; WANGENHEIM, C. G.; ZOUCAS, A. Implementação do MPS.BR em grupo de empresas da ACATE em Florianópolis 2007/2008. *III Workshop de Implementadores (W2 – MPS.BR)*. Belo Horizonte (MG), nov., 2007.

VERMEULEN, A.; AMBLER, S. W.; BUMGARDNER, G.; METZ, E.; MISFELDT, T.; SHUR, J. *The elements of Java style*. USA: Cambridge University Press, 2000.

WHITTAKER, J. A. What is software testing? And why is it so hard? *IEEE Software*, vol. 17, issue 1, pp. 70-79, Jan./Feb., 2000.

WILLIAMS, L.; COCKBURN, A. Agile software development: it's about feedback and change. *IEEE Software*, vol. 36, issue 6, pp. 39-43, Jun., 2003.

YOSHIDA, D.; KOHAN, S.; SALVETTI, N.; HIRAMA, K. Convênio com qualidade: implementação MPS.BR nível G em grupo de empresas de São Paulo. *III Workshop de Implementadores (W2 – MPS.BR)*. Belo Horizonte (MG), nov., 2007.