# ARQUITETURA DE SISTEMAS

AUTOR
ALEXANDRE MÁRCIO MELO DA SILVA

# ARQUITETURA DE SISTEMAS

ALEXANDRE MÁRCIO MELO DA SILVA

1ª EDIÇÃO
SESES
RIO DE JANEIRO 2017

**Conselho editorial** ROBERTO PAES E LUCIANA VARGA

**Autores do original** ALEXANDRE MÁRCIO MELO DA SILVA

**Projeto editorial** ROBERTO PAES

**Coordenação de produção** LUCIANA VARGA, PAULA R. DE A. MACHADO E ALINE KARINA RABELLO

**Projeto gráfico** PAULO VITOR BASTOS

**Diagramação** LUÍS SALGUEIRO

**Revisão linguística** BERNARDO MONTEIRO

**Revisão de conteúdo** CLEBER GRAFIETTI

**Imagem de capa** ST22STUDIO | SHUTTERSTOCK.COM



Dados Internacionais de Catalogação na Publicação (CIP)

S586a Silva, Alexandre Márcio Melo da
Arquitetura de sistemas / Alexandre Márcio Melo da Silva.
Rio de Janeiro : SESES, 2017.
144 p.
ISBN: 978-85-5548-477-3.

1. Arquitetura de sistemas. 2. Interação de componentes.
3. Workflow. 4. UML. I. SESES. II. Estácio

CDD 658.403

Diretoria de Ensino — Fábrica de Conhecimento
Rua do Bispo, 83, bloco F, Campus João Uchôa
Rio Comprido — Rio de Janeiro — RJ — CEP 20261-063

# Sumário

# Prefácio

Prezados(as) alunos(as),

É importante compreendermos que a arquitetura de sistemas é fundamental para o desenvolvimento do software, pois é através dela que se torna possível representar os componentes do mesmo, os seus relacionamentos com o ambiente e entre si, além de auxiliar em seu design e evolução. E, dentro do processo de desenvolvimento de sistema, a arquitetura é responsável por orientar a forma como os requisitos serão implantados e a maneira como o desenvolvimento irá prosseguir, utilizando-a como solução para determinado problema. Outro ponto importante que devemos compreender é que a arquitetura de sistemas é uma excelente ferramenta de comunicação, pois é através dela que a integridade conceitual do projeto é comunicada entre as partes interessadas do projeto e entendida por eles.

Dessa forma, podemos ressaltar que ela é uma forma de apresentar soluções a determinados problemas, adequando a solução em um padrão que possa ser entendido tanto por técnicos, gerentes e desenvolvedores, quanto por indivíduos que possuem nenhum conhecimento em desenvolvimento de sistemas. É uma maneira de permitir um maior entendimento e análise da ferramenta que se pretende desenvolver no projeto sem de fato levarmos em consideração que a arquitetura é uma boa prática de desenvolvimento e que, sem ela, é muito provável que o projeto que não a utilizar venha a se tratar de um projeto que dificilmente atenderá a todos ou somente aos requisitos mais importantes. Em suma, sem a arquitetura de sistemas, esse será um projeto no qual nem todos saberão o que realmente deverá ser feito em todos os seus detalhes, gerando bastante estresse e conflito entre os envolvidos justamente por não chegarem e não possuírem um modelo que represente o senso comum do que será desenvolvido. Dificilmente um projeto sem um bom planejamento e sem a arquitetura de sistemas irá seguir de fato um cronograma e orçamento definidos, o que mais cedo ou mais tarde levará ao fracasso do projeto.

Portanto, devemos termos em mente a importância de se utilizar boas práticas de desenvolvimento de software, como a arquitetura de sistemas. Compreender seus detalhes como componentes de uma arquitetura, o que eles são, como se relacionam, como são classificados, quais são as suas características, como modelar uma arquitetura, o que é interface, como especificá-la, como defini-las,

entre outras coisas, está dentro do planejamento deste livro. Bem como o processo de desenvolvimento RUP e o padrão arquitetural MVC, além do framework para arquitetura de sistemas distribuídos, o CCM.

Diante de todos os aspectos descritos acima, este livro foi desenvolvido para que você, aluno, possa compreender melhor a importância desta disciplina para a sua carreira, seja ela acadêmica ou profissional. Esperamos que, com um estudo dedicado e atencioso a este material, você, aluno, desperte o seu interesse pelo desenvolvimento de sistemas através do uso de boas práticas de desenvolvimento.

**Bons estudos!**

# 1

# Componentes de sistemas e o processo de desenvolvimento

# Componentes de sistemas e o processo de desenvolvimento

## Introdução

Devemos ter em mente que não é fácil desenvolver um *software*. E é justamente por isso que vários projetos nessa área fracassam durante seu desenvolvimento ou ao alcançar os seus resultados. E, quando nos referimos a fracasso, implicitamente estamos nos referindo, também, aos altos custos no orçamento que isso gera. Definitivamente, eles não solucionam os problemas da forma que deveriam. Alcançar um bom produto de *software* está relacionado à sua complexidade e, também, à complexidade do seu processo de desenvolvimento.

Portanto, aprenderemos como a arquitetura de sistemas pode nos ajudar a alcançar bons resultados através do uso de modelos padronizados que garantem tanto o desenvolvimento de *software* quanto a gerência de processos de desenvolvimento do mesmo.

Diante disso, neste capítulo estudaremos pontos relevantes para o entendimento de arquitetura de sistemas, de componentes e de processo de desenvolvimento. Formaremos, assim, uma base de conhecimento para amenizar riscos no desenvolvimento de sistemas.

## OBJETIVOS

- Aprender os principais conceitos de componentes de sistemas;
- Conhecer os conceitos sobre arquitetura de sistemas e a sua importância;
- Entender a importância de aplicar o conceito de arquitetura a componentes;
- Poder identificar os componentes de sistemas no seu dia a dia;
- Conhecer e aprender o padrão estrutural em camadas;
- Aprender os fundamentos de *workflow*;
- Compreender os conceitos de processo de desenvolvimento de sistemas;
- Entender a aplicabilidade de *workflow* em modelos de gestão;
- Compreender os objetivos de metodologia de desenvolvimento e de gestão.

## Componentes

Um dos conceitos sobre componentes de sistemas o relaciona com o fato de eles poderem ser objetos ou um conjunto dele, sistemas ou alguma implementação, desde que esse componente seja **autossuficiente** e **independente**. Uma outra definição é dada por Brown e Wallnau (1996): "uma não-trivial, quase independente, **e substituível** parte de um sistema que cumpre uma função clara no contexto de uma arquitetura bem definida". Krutchen (1998) reforça a ideia de o componente ser independente e substituível, porém ele possui a sua importância dentro de um contexto com uma função clara que foi definida anteriormente. Gimenes (2005), por sua vez, aborda os componentes como sendo **unidades independentes de *software***, capazes de encapsular dentro de si o projeto e a implementação. Além de, por intermédio de suas interfaces, oferecerem serviços ao ambiente externo.

Os vários conceitos existentes sobre componentes de sistemas convergem principalmente para o fato de eles serem independentes e substituíveis. Dessa forma, o componente deve ser separado/independente do meio em que está inserido e dos demais componentes que possam existir. Além de o componente não possuir "contato" com os demais, ele não deve permitir o acesso a seus detalhes de construção, o que nos leva ao conceito de Gimenes (2005) quando a autora aborda que os componentes devem encapsular, no caso, a sua implementação. Mas o componente não é apenas uma unidade inserida em um sistema de forma isolada. Como citado por Brown e Wallnau (1996), além de Krutchen (1998), ele está ali pois a ele foi dada uma funcionalidade, possuindo, assim, a sua importância em determinado contexto. Somado a isso, para que a sua funcionalidade possa ser aproveitada, os componentes interagem com o ambiente e outros componentes por intermédio de interfaces, e não diretamente.

Sendo assim, outra definição de componente de *software* o trata como unidade de composição com interfaces, onde essas interfaces são bem definidas e implantadas de forma independente mas podem ter essa composição, ou seja, podem ser combinadas com outras partes (Szyperski, 2002). Essas interfaces separam a especificação do componente de sua implementação. Dessa forma, elas não permitem que os usuários tenham acesso aos detalhes da implementação do componente. Mas a sua especificação é publicada de forma separada do código fonte.

Os componentes possuem objetivos específicos dentro de um contexto, e os seus conceitos assemelham-se com o conceito de objetos em POO. Na Programação Orientada a Objetos, chamamos de componentes a classe responsável por implementar uma interface e que possua autonomia em relação aos outros. Vale ressaltar que objetos não implementam interfaces. O objeto é, nesse contexto, a instância de uma classe que é responsável por implementar uma interface.

O usuário ou cliente pode construir o próprio componente ou adquirir os componentes COTS (*Commercial Off-The-Shelf*), que são componentes com domínio específico. Mais genéricos, eles podem ser adquiridos comercialmente: são os chamados componentes de prateleira. Porém, o seu uso de forma efetiva necessita de desenvolvimento de métodos com maior precisão para produção e utilização (Carney, 1997).

Cheesman (2001) pontuou os princípios fundamentais e conceituais acerca de componentes de sistemas. Eles são os seguintes:

- **Unificação dos dados e da função:** um componente é composto de valores, que podem ser de dados ou estados, e das funções que são responsáveis pelo processamento de tais dados ou estados. Dessa forma, podemos afirmar que existe uma dependência natural entre os dados ou estados e a função;
- **Encapsulamento:** o fato de como os componentes armazenam os dados ou estados, ou, ainda, como as funções que possuem são executadas, é algo que não faz parte do conhecimento do usuário do componente. Esse usuário ou cliente não depende da implementação de um componente, apenas da especificação do mesmo. O encapsulamento diminui a dependência e o acoplamento entre vários componentes e o sistema;
- **Identidade:** em um sistema, podem existir vários componentes, e cada um deles possui uma identidade única.

Como já foi citado, os componentes são substituíveis e independentes; sendo assim, eles podem ser reutilizados. E, se existir dependências entre componentes, eles deverão ser tidos com um componente reutilizável. Geralmente, esses componentes estão em um único local do sistema, e não espalhados ou misturados com outros artefatos. E também, como já citado, eles interagem entre si com o uso de interfaces: essa interação se dá por meio de conexão entre um componente e outro componente ou sistema, não existindo a necessidade dos detalhes de como essa comunicação é realizada estarem explícitos. Um fator essencial para que os

componentes possam atingir os seus objetivos é uma documentação adequada e clara dele.

Sametinger (1997) pontuou algumas características dos componentes:

- **Funcionalidade:** indica a aplicabilidade do componente dentro de um determinado sistema, o que facilita o seu reuso;
- **Interatividade:** leva em consideração as diferentes formas de comportamento da interação existente dos componentes entre si e no meio em que estão inseridos;
- **Interação:** informa como ocorre a interação dos componentes entre si e seus usuários;
- **Concorrência:** cuida da concorrência entre os componentes e os fatores ligados para essa verificação;
- **Distribuição:** leva em consideração os aspectos que são necessários para que seja permitida a troca de dados e a manutenção da comunicação na distribuição dos componentes;
- **Formas de adaptação:** o cuidado na adaptação do componente ao longo do tempo com o intuito de que ele continue sendo reutilizável;
- **Controle de qualidade:** cuidado na avaliação e na garantia da qualidade do componente.

Em resumo, os componentes de *software* ou sistema são unidades de *software* que possuem funcionalidades que são responsáveis por executar atividades definidas no sistema. **São desenvolvidos e implantados para que atuem de forma independente com a possibilidade de serem combinados** entre si, por meio de interfaces, para formar um sistema. A interface de um componente possui a responsabilidade de definir como os outros componentes ou sistema podem solicitar as funcionalidades ou serviços do componente. Os componentes possuem a capacidade de serem configurados de acordo com a especificação dos usuários ou clientes. E, por não possuírem dependência com o sistema, são reutilizáveis e facilmente atualizados. Somado a isso, os componentes, em geral, são distribuídos em código binário e criados utilizando métodos e técnicas, além de ainda ferramentas padronizadas de desenvolvimento.

## Exemplos de componentes de sistema

Para exemplificar (Figura 1) e fixarmos melhor o conceito de componentes de sistemas, suponhamos que temos um motor com duas entradas (inicia e para) que coletam informações de dois botões. E, para que haja essa coleta de informações, ambos estão conectados. Sendo assim, o motor possui duas entradas que recebem informações de dois botões. O motor também está conectado a um medidor por meio de uma saída (velocidade). A conexão existente entre os dois botões e o motor apresenta uma interface ou conector que recebe uma ocorrência gerada por um componente (um dos botões). Tal ocorrência ativa um método (inicia ou para) presente em outro componente (motor). A conexão entre o motor e o medidor apresenta um tipo de interface ou conector em que um par de valores é sincronizado de forma contínua.

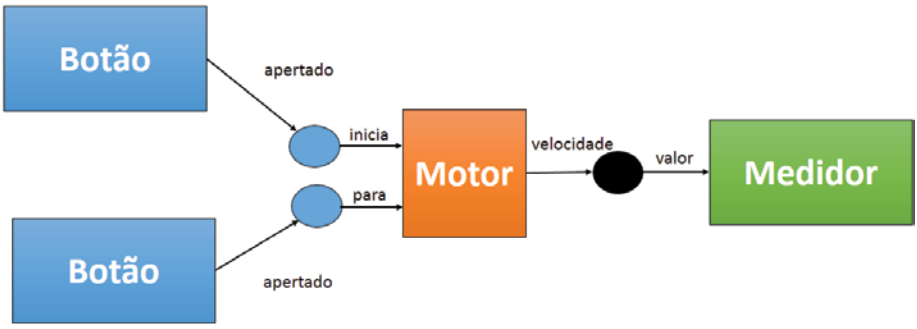


Figura 1. Exemplo de um kit de componentes. Adaptado de D'Sousa (1998).

Observe que, nesse exemplo, temos quatro componentes: os dois botões, o motor e o medidor. Chamamo-los de componentes, pois eles se adequam ao seu conceito. Observe que cada um tem suas funcionalidades e que eles interagem entre si por meio de interfaces (os círculos), e não diretamente. Observe, também, que esses componentes podem ser reutilizados em outros sistemas: por exemplo, os botões podem representar ações como liga e desliga em outro contexto, e o medidor pode representar outros valores além da velocidade.

Um segundo exemplo (Figura 2) de componentes de sistema, que foi exemplificado por D'souza e Wills (1998), trata de um sistema que faz leitura

de dados a partir de uma fonte (News Feed), os registra em Planilha e repassa os resultados para um Banco de Dados. Esse último componente é compartilhado também com uma Aplicação Web que coleta as informações sob demanda. Cada um desses três componentes (News Feed, Planilha e Banco de Dados) é, de certa forma, independente, compartilhado e utilizado. Juntos, esses componentes formam um sistema maior. Observe que eles possuem suas interfaces para suas respectivas conexões, e, como estudado anteriormente, isso permite que sejam facilmente substituídos por outros componentes equivalentes.

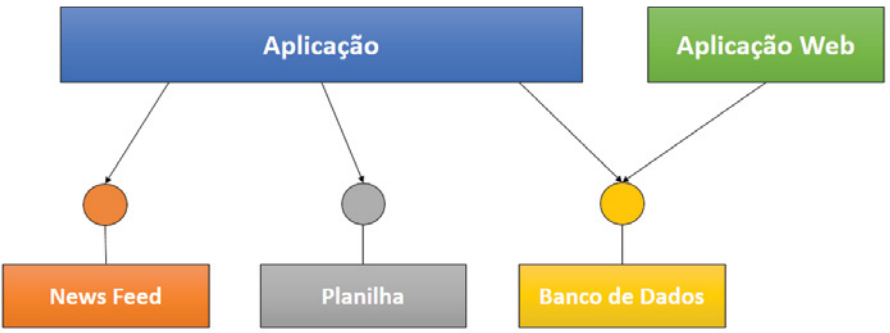


Figura 2. Exemplo de uso de componentes de sistema. Adaptado de D'Souza e Wills (1998).

Outro exemplo de componentes que podemos citar são os *widgets*, que são componentes visuais de interface com os usuários. Em IDEs (Ambiente de Desenvolvimento Integrado), é disponibilizado ao programadores uma coleção de componentes visuais para que sejam utilizados na composição de interface com o usuários. Basicamente são caixas que o programador arrasta para montar uma interface posicionando os componentes, os instanciando, configurando as suas propriedades e fazendo associações de métodos a seus eventos (como clique de botões, por exemplo), sem ter o acesso ao código fonte desses componentes. Os CMS (Sistema Gerenciador de Conteúdo) também apresentam muito esse conceito com o uso de formulários, sistemas de comentário, componentes de galerias de fotos etc.

## Arquitetura de sistemas

Diversas são as soluções que podem ser identificadas quando procuramos solucionar determinado problema. Mas fatores como, por exemplo, o custo

possuem forte influência na escolha de qual solução identificada será adotada. O mesmo se repete no que diz respeito ao desenvolvimento de *softwares*. Podemos encontrar várias soluções computacionais com potencial para atender aos requisitos, e para minerar essas soluções se faz necessário o uso de análise para definir a solução que mais se adequa ao contexto de desenvolvimento da aplicação.

A arquitetura de sistemas ou *software* é a abordagem mais **utilizada para representar soluções**; dessa forma, se alcança a solução mais adequada ao problema. A arquitetura é baseada em modelo de alto nível para permitir um maior entendimento e uma análise mais fácil do *software* que se pretende desenvolver.

Garlan e Perry (1995), além de Kazman (2001), apontam duas tendências para o uso de arquitetura na representação de soluções de *software*:

- O reconhecimento de que esse tipo de abstração permite a visualização e o entendimento de determinadas propriedades pelos projetistas de *software*;
- A diminuição de esforço no desenvolvimento de produtos por meio da integração de partes já desenvolvidas graças aos frameworks.

Vale ressaltar que a arquitetura não serve apenas para facilitar o entendimento dos projetistas mas também para o entendimento dos vários *stakeholders* que participam do projeto de software e de seu desenvolvimento (GARLAN, 2000). Em geral, a arquitetura possibilita o seu uso como uma **ferramenta para comunicar a solução proposta a todos os envolvidos**; ela, por ser um modelo de alto nível, facilita o entendimento de quem não possui os conhecimentos técnicos. Para que haja essa comunicação da solução, é desenvolvido um artefato ou documento chamado de documento arquitetural.

E, logicamente, se a arquitetura está sendo utilizada para verificar a solução mais adequada para um problema, ou seja, a solução que deve resolver determinado problema, e esse problema é descrito em forma de requisitos, tanto a arquitetura quanto o documento arquitetural devem utilizar os requisitos como a principal fonte de informações. Veja bem, os requisitos devem ser utilizados como a principal fonte; portanto, existem outras fontes que podem ser utilizadas para definir os elementos arquiteturais e como eles serão organizados. Nessas outras fontes, as que possuem um maior destaque são: experiência do arquiteto, raciocínio sobre os requisitos, estilos e táticas arquiteturais.

Segundo Bass e colaboradores (2003), a arquitetura de *software* é responsável por representar a estrutura ou o conjunto dela, compreendendo os elementos de

*software* bem como as suas propriedades externas e relacionamentos. Mas, para que se possa criar tal estrutura, alguns elementos básicos, defendidos por Dias e Vieira (2000), podem ser utilizados:

- **Elementos de *software:*** esses são os componentes, podendo também ser chamados de módulos. São as abstrações que possuem como responsabilidade representar as entidades, que, por sua vez, implementam funcionalidades bem definidas;
- **Conectores:** são as interfaces, podendo também ser chamadas de relacionamentos. São as abstrações que possuem como responsabilidade representar as entidades, que, por sua vez, facilitam a comunicação entre os elementos de *software*;
- **Organização ou configuração:** é basicamente a forma como estão distribuídos e organizados os elementos de *softwares* e seus conectores.

Possuindo essa estrutura, ela deve ser representada na forma do documento arquitetural que, como já citado, facilitará o entendimento para que a arquitetura (solução) seja utilizada para seu devido fim da forma que foi projetada. O que compõe o documento arquitetural é uma série de modelos e informações que possibilitam a descrição da estrutura projetada para que os requisitos sejam atendidos. Esse documento também pode ser chamado de Documento de Arquitetura de *Software*.

A arquitetura de sistemas é desenvolvida nos estágios iniciais do processo de desenvolvimento de *software* justamente por permitir a definição e visualização da solução que se pretende criar. Essa arquitetura inicial é utilizada pelos *stakeholders* para que a possam utilizar como base na tomada de decisões. Sabendo desses fatores, é compreensível o porquê do documento arquitetural pertencer ao escopo. E, como sabemos, o escopo pode sofrer alterações no decorrer do processo de desenvolvimento de *software*; então, o mesmo pode ocorrer com o documento arquitetural. Segundo Clements e colaboradores (2004), a arquitetura inicial vai sofrendo refinamentos no decorrer do desenvolvimento de *software*, e isso reduz o nível de abstração, permitindo, por exemplo, que a representação dos conectores entre os elementos de *software* e os arquivos de código fonte que os implementam possam ser visualizados. Dessa forma, a arquitetura passa a ser utilizada na tomada de decisão relacionada à qual ou às quais tecnologias se deverá utilizar

para a implementação da solução, pois ela passa a incorporar as informações relacionadas às decisões do projeto.

Vale ressaltar que não existe uma forma única de construir uma arquitetura de *software* e o documento arquitetural. No que diz respeito à arquitetura de *software*, listamos alguns dos diversos padrões existentes:

- **Estrutura:** Camadas, Pipes e Filtros, Quadro-negro;
- **Sistemas adaptáveis:** Reflexo, Microkernel;
- **Sistemas distribuídos:** Broker;
- **Sistemas interativos:** Modelo - Visão - Controle (MVC), Apresentação - abstração - Controle.

Observe que separamos os padrões em algumas categorias; isso quer dizer que, de acordo com o contexto e a necessidade, o nosso sistema, ou parte dele, poderá ser projetado em várias formas de arquitetura. Neste capítulo, trataremos do padrão estrutural Camadas, e no decorrer deste livro iremos tratar o padrão de sistemas interativos MVC.

## Camadas da arquitetura de sistemas

A arquitetura de *software* em camadas é utilizada no contexto em que ela envolve um sistema de grande porte que deve ser decomposto para que algumas questões possam ser resolvidas em vários níveis de abstração. Dessa forma, os sistemas devem ser estruturados agrupando os componentes em forma de camadas, ou seja, um sobre os outros. As camadas tidas como superiores devem fazer uso dos serviços providos pelas camadas que estão abaixo delas. Uma camada não poderá utilizar algum serviço que seja fornecido pelas camadas de cima. Dessa mesma forma, não se deve pular camadas, a não ser que existam camadas intermediárias que apenas adicionam componentes de acesso.

Cada camada nesse padrão de arquitetura deve representar um grupo de funcionalidades, sendo assim:

- Nas camadas superiores está a funcionalidade específica do sistema;
- Nas camadas intermediárias está a funcionalidade que envolve os domínios de negócio do sistema;
- Nas camadas inferiores está a funcionalidade específica do ambiente onde o sistema foi implantado.

Geralmente, é criada apenas uma camada responsável pela funcionalidade específica do sistema. Quando o sistema complexo, ou de grande porte, é composto por outros sistemas menores, as camadas intermediárias responsáveis pelo domínio de negócio devem estar estruturada em camadas. E, por fim, as camadas inferiores têm um papel fundamental, em que possivelmente poderão ter várias camadas para que o sistema funcione adequadamente no local onde for implantado.

Na figura a seguir, temos um exemplo de arquitetura com quatro camadas. Analisaremos cada uma para melhor entendimento sobre esse tópico.


Figura 3. Arquitetura de software quatro camadas.

A camada Subsistemas de Aplicativos possui os serviços e funcionalidades específicas do software que agrega valor à organização. A camada Específico do Negócio possui os componentes que são específicos do tipo de negócio de que trata o sistema. Na camada *Middleware*, há os componentes que são responsáveis pela interface com o usuário (GUI), a interface que conecta o sistema com o seu banco de dados, os serviços fornecidos pelo sistema operacional independente de plataforma e os componentes, como, por exemplo, planilhas, editores de textos, editores de diagramas etc. E, por fim, a camada *Software* de Sistema é composta pelos componentes sistemas operacionais, banco de dados, interfaces da aplicação com algum *hardware* em específico etc.

## Arquitetura de componentes

Segundo Wallnau (2001), existe uma conformidade entre a arquitetura de *software* e os componentes, pois, entre outros fatores, ambos são centralizados na qualidade dos atributos e na simplificação no que diz respeito à integração entre as partes do sistema. Através dessa conformidade, é possível verificar o papel fundamental da arquitetura de *software* para os componentes. Pois, se utilizando da arquitetura, é possível especificar a intrerconexão existente entre os componentes de um sistema.

A partir da necessidade de melhorar o processo de desenvolvimento de *software*, e procurando aumentar a produtividade, a qualidade e a redução de custos, além de novas concepções para o desenvolvimento, surgiu o desenvolvimento de sistemas baseados em componentes (VINCENZI, 2005). Com esse tipo de desenvolvimento, os sistemas passaram a ser desenvolvidos em diversas partes, cada uma com sua funcionalidade bem definida, e não mais como um único módulo. Porém, segundo Werner (2000), os componentes não podem ser tidos como totalmente independentes, de forma isolada entre si e com relação ao seu ambiente. Desse modo, a importância da arquitetura de *software* é reforçada, pois por meio dela é possível detalhar de forma melhor a interconexão entre os componentes.

Listamos algumas vantagens dos componentes:

- **Reusabilidade:** componentes podem ser reutilizados em diversos sistemas, podendo até desenvolver *softwares* completos apenas reutilizando componentes;
- **Produtividade:** é possível desenvolver sistemas com maior rapidez e simplicidade;
- **Facilidade de uso e aprendizado:** não há necessidade de treinamento extenso para a utilização de componentes;
- **Mudanças executadas com facilidade e rapidez:** os componentes, por não possuírem dependências e pela sua característica de serem modulares, permitem que alterações sejam aplicadas com a mesma rapidez com que os negócios mudam;
- **Melhor foco de negócios:** devido ao alto nível de abstração dos componentes, é possível aos gerentes e desenvolvedores trabalharem com foco nos negócios em alto nível.

Gimenes (2005) apontou alguns outros benefícios que estão relacionados com componentes e o fato de eles serem reutilizáveis:

- **Gerenciamento de complexidade:** o gerenciamento de complexidade é reduzido por ser tratado um número cada vez menor de componentes por vez. E, desse modo, também diminuem os riscos de desenvolvimento;
- **Desenvolvimento paralelo:** os componentes possuem independência e permitem que os desenvolvedores concentrem-se apenas nas interfaces entre os componentes;
- **Aumento de qualidade:** como os componentes são reutilizáveis, eles podem já ter sido utilizados e testados em outros ambientes;
- **Facilidade de manutenção e atualização:** justamente por ser desenvolvido com componentes, o sistema facilita a localização para alterações e atualizações do mesmo.

No tocante a dificuldades e problemas enfrentados pelo uso de desenvolvimento baseado em componentes, segundo Gimenes (2005), são citados:

- **Seleção do componente certo:** selecionar o componente certo é uma tarefa árdua, pois pesquisas na organização e recuperação de bibliotecas de componentes precisam ser realizadas;
- **Confiabilidade dos componentes:** deve ser fornecida uma documentação que assegure que foi realizada uma série de testes em cada um dos componentes em ambiente compatível;
- **Custo e tempo de desenvolvimento:** a reutilização de componentes pode necessitar de esforço extra em reutilizações futuras, precisando em seu desenvolvimento ser mais flexível, estável e correto. E, muitas das vezes, esse esforço de tempo e custo pode ser superior ao desenvolvimento de um sistema com finalidade específica;
- **Cultura dos engenheiros de *software*:** por muitas vezes, os engenheiros de *software* querem ter total controle sobre o desenvolvimento e acabam relutando em utilizar e confiar em códigos de terceiros, pois, geralmente, gostam de soluções próprias.

## Camadas de arquitetura de componentes

Na figura a seguir, temos um exemplo de arquitetura de componentes com camadas em que analisaremos cada uma. A primeira camada, a superior, é responsável por controlar a maneira como os dados são apresentados e a coleta da informação passada pelo usuário através do uso da interface do sistema. A segunda camada, a intermediária, tem a responsabilidade de armazenar os estados dos componentes; esses estados são resultados dos eventos que são coletados. A

última camada, a inferior, tem a responsabilidade de permitir que os eventos sejam distribuídos na rede de computadores; dessa forma, comunicam os componentes que estão executando em várias estações de trabalho.

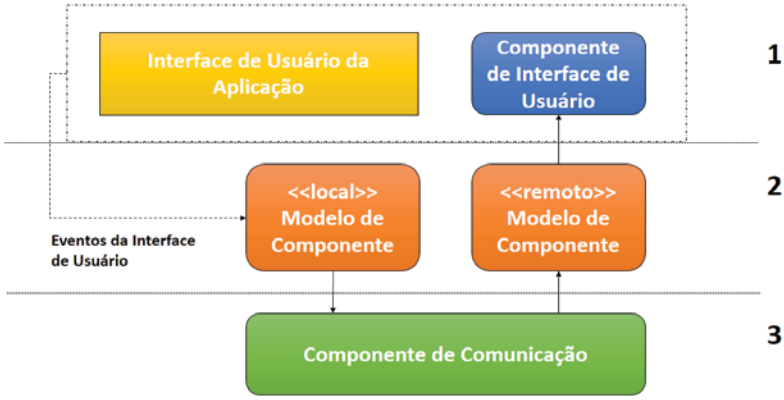


Figura 4. Visão geral da arquitetura de componentes.

O modelo utilizado na arquitetura de componentes, representado pela Figura 4, representa um sistema que monitora os eventos realizados na interface do usuário, na qual esses eventos são coletados e armazenados por componentes. A sincronização entre os dados e suas cópias remotas, e as sessões de trabalho, são de responsabilidade da camada de comunicação. Nela, esses dados são apresentados ao usuário em cada cópia remota. E, assim, todos os usuários espalhados pela rede podem visualizar os dados.

## Modelos de componentes

Basicamente, os **modelos de componentes servem para mostrar como os componentes estão organizados e como está a questão da dependência entre eles**. Componentes de outros *frameworks* podem ser reutilizados nos modelos. Um modelo de componentes é responsável pela definição de diversos fatores relacionados com o desenvolvimento e interação dos componentes. Entre os quais, fazem parte: especificar componente, instanciar componente, quais tipos de interfaces e portas liberadas, o modelo de dados padrão, como as ligações são realizadas, como as transações são distribuídas, tipos de interfaces, interfaces obrigatórias, catálogo e localização dos componentes, emissão de requisições e respostas, segurança, repositórios, formato, metadados,

interoperabilidade, documentação, nomeação, mecanismos de customização, composição, evolução, controle de versões, instalação e desinstalação, serviços de execução, empacotamento, entre outros (HEINEMAN e COUNCILL, 2001; D'SOUZA e WILLS, 1998).

Weinreich e Sametinger (2001) definem modelo de componentes como **o padrão de descrição de interface, definindo quais são os padrões de composição.** O tipo mais comum de acoplamento entre dois componentes são publicador/ouvinte e o cliente/servidor. No primeiro, o ouvinte faz seu registro como tratador de eventos e informações que são publicadas pelo publicador. No segundo, o cliente faz chamadas ao servidor. Também faz parte do modelo de componentes definir os tipos de interfaces disponibilizadas. Esse modelo define o padrão de *deployment*, especificando a estrutura e semântica que os arquivos descritores devem possuir (WEINREICH e SAMETINGER, 2001).

A Figura 5 representa, para melhor entendimento dos conceitos aqui tratados, um projeto baseado em componentes, relacionando componentes, modelo de componentes e *frameworks* de componentes.

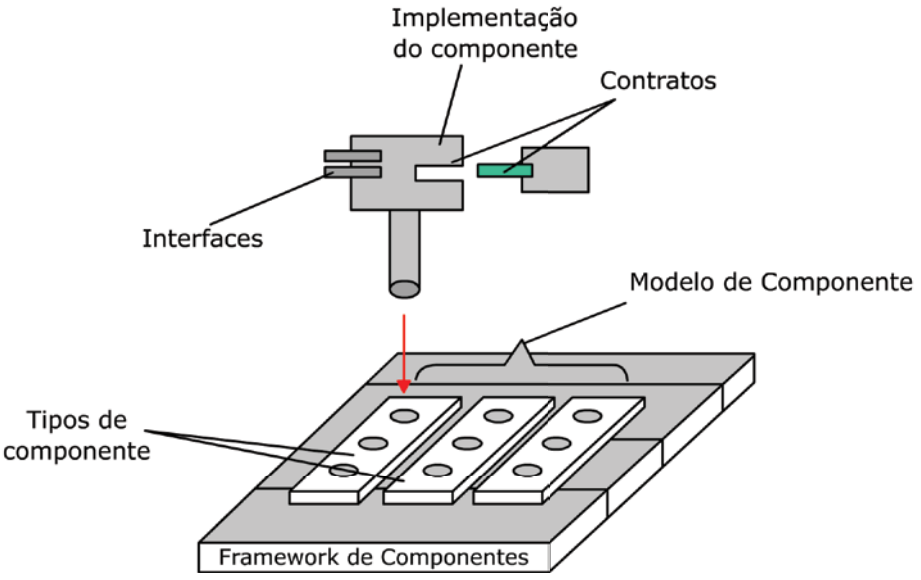


Figura 5. Padrões de projetos baseados em componentes. Adaptada de Bachman (2000).

Os componentes são administrados por contratos que garantem que os componentes obedecem determinadas regras. Diferentes tipos de componentes podem existir, representando cada um o seu papel no sistema que é descrito por uma interface. O modelo de componente contém os tipos de componentes, suas interfaces e especificações de padrão de interação. E o *framework* de componentes dá todo o suporte e reforço necessários ao modelo de componentes.

Ainda não existe um consenso sobre o que deve constar ou não em um modelo de componentes; Bachmann (2000) e Spagnoli (2003), por exemplo, defendem os seguintes padrões e convenções:

- **Tipos de componentes:** a definição dos componentes leva em consideração a função de seus conectores, que, por sua vez, os relaciona a um tipo. Por exemplo, um componente implementa os conectores F, G e H; então, ele é do tipo, F, G e H, e pode executar o papel de F, G ou H em momentos diferentes;
- **Formas de interação:** modelos de componentes devem especificar como é a interação dos componentes com outros ou com um *framework* de componentes. A interação entre componentes envolve restrições, como: tipos de componentes, número permitido de clientes simultâneos, qual tipo de cliente é permitido para determinado tipo, entre outros. Já a interação entre componentes e *frameworks* inclui: operações de gerenciamento de recursos, ativação e desativação de componentes, persistência, entre outros;
- **Definição de recursos:** modelos de componentes devem descrever quais são os recursos permitidos a cada componente, como e quando será possível ligar tais recursos.

### Exemplos de modelos de componentes

A Microsoft propôs um modelo de componentes chamado de **OLE** (*Object Linking and Embedding*) cujo propósito era fazer a integração de diversos objetos que eram gerados por vários programas para um único documento. Quando aplicações compatíveis como o modelo são instaladas, elas ficam disponíveis para disponibilizar e receber conteúdos. Por exemplo, o Microsoft Word pode servir de container, carregando todos os conteúdos que são disponibilizados por outros programas. Outros modelos que foram utilizados pela Microsoft a partir do OLE são: **COM, DCOM e o ActiveX**. Esses modelos oferecem suporte para vários níveis de complexidade e possibilitam que haja a integração de programas

desenvolvidos por vários fabricantes. O foco da solução trata de componentes binários interoperáveis.

O **JavaBeans** é um outro modelo de componentes, mas esse foi fabricado pela Sun. Ele define como serão tratados eventos, propriedades, introspecção, reflexão, customização e empacotamento de componentes. O **Enterprise JavaBeans** já é um modelo voltado para sistemas corporativos, tratando da interconexão de componentes remotos.

Um outro exemplo de modelos de componentes é o **Web Services**, que é um padrão de comunicação entre componentes através da internet. Um sistema pode fazer o uso de serviços de componentes por meio do protocolo SOAP, que, por sua vez, encapsula as chamadas e os retornos dos serviços em pacotes no formato XML.

Um último exemplo é o **CORBA** (*Common Object Request Broker Architecture*), e trataremos dele com mais detalhes em outro ponto deste livro. Ele fornece um modelo de componentes que permite a comunicação entre componentes distribuídos, utilizando IDL (Interface Definition Language) com o intuito de descrever interface pública de seus serviços. Dessa forma, o cliente não possui dependência do local onde se encontra o objeto que está sendo executado nem da linguagem na qual ele foi programado ou do sistema operacional que está sendo utilizado.

## Workflow

O termo "workflow" é bastante citado com o intuito de significar "processos de negócio" ou um conjunto de atividades que interagem entre si com o intuito de atingir um objetivo. Existe uma entidade denominada WfMC (*Workflow Management Coalition*) que é responsável por disseminar o uso de tecnologias de *workflow*, e ela faz isso através do desenvolvimento de padrões e terminologias. E, para o *Workflow Management Coalition* (1996), *workflow* representa a **automação de processo de negócio, em que documentos, informações ou tarefas são distribuídos de acordo com procedimentos predefinidos**. Um processo de negócio é constituído de uma série de atividades ou tarefas que se relacionam com o objetivo de alcançar as metas de negócio no que diz respeito à estrutura organizacional (*Workflow Management Coalition*, 1996). E, para tal, a *WfMC* criou o *Workflow Reference Model*, que é um modelo voltado para sistemas de *workflow*, buscando identificar características, termos e componentes; adicionalmente a isso, ele habilita especificações para o desenvolvimento dentro desse ambiente.

Na Figura 6, podemos observar as terminologias básicas que, segundo o *Workflow Management Coalition* (1996), estão relacionados ao *workflow*. Elas são descritas logo a seguir:

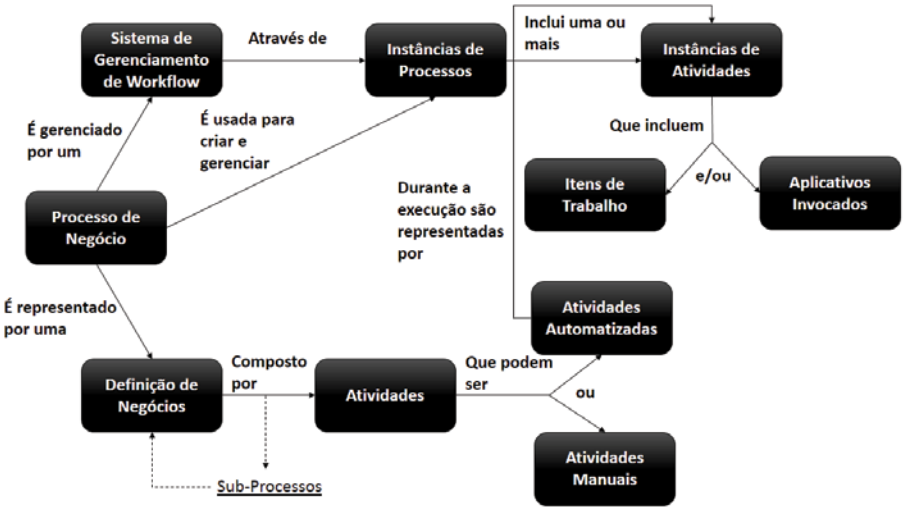


Figura 6. Terminologia básica de workflow. Adaptado de Workflow Management Coalition (1999).

- **Processo de Negócio:** O que é esperado que ocorra. Possui uma ou mais diretrizes relacionadas ou atividades que, em conjunto, atendem a um objetivo de negócio; regras e relacionamentos são definidos para o seu devido funcionamento;
- **Definição de Processos:** Uma representação do que é esperado que ocorra, a representação de processos;
- **Sistema de Gerenciamento de *Workflow*:** Controla aspectos automatizados do processo de negócios. É um sistema que, através da execução de um programa, é capaz de definir, gerenciar e controlar o *workflow*;
- **Instância de Processo:** Uma representação do que realmente acontece, a execução de processo;
- **Atividades:** Formam passo lógico que pertence a um processo, descrevendo formalmente uma parte de trabalho;
- **Atividades Manuais:** Atividades que não são gerenciadas pelo *workflow*;
- **Instâncias de Atividade:** Relaciona a execução da atividade a uma instância única de processo;

• **Itens de Trabalho:** Representa as tarefas que devem ser executadas por participante do *workflow* dentro do contexto de atividade e instância de processo;

• **Aplicativos Invocados:** Ferramentas e aplicativos utilizados para a execução de uma atividade.

## Gerenciamento de processos

Segundo Pressman (2006), processos de *software* formam o alicerce no que diz respeito ao controle gerencial de projetos de software, além de estabelecer o meio para a aplicação de métodos técnicos, a produção de produtos de trabalho, estabelecimento de metas, assegurar a qualidade e a gerência adequada de alterações no projeto. E, para se ter um controle gerencial, métodos devem ser utilizados, e esses métodos dão um direcionamento em "como fazer" para se desenvolver um *software*. E esses métodos ou metodologias são compostos por uma série de tarefas que envolvem comunicação, análise de requisitos, modelagem de projetos, construção de *softwares*, testes e manutenção.

Sommerville (2007) retrata o processo de *software* como um conjunto bem estruturado de atividades, em que essas atividades têm um papel fundamental para o desenvolvimento de *software*. É fato que existem diversos processos de *software*, porém todos eles possuem atividades comuns. Sommerville (2007) as destaca:

• **Especificação:** os requisitos são especificados, apontando quais são as funcionalidades e restrições que o *software* deve possuir;

• **Projeto de implementação:** o projeto do *software* é realizado, e o mesmo passa a ser codificado de acordo com o projeto e suas futuras modificações;

• **Validação:** ajuda a garantir que o que está sendo implementado é de fato o que foi especificado, a garantia de atender à necessidade do cliente;

• **Evolução:** trata da resposta dada às modificações de acordo com a necessidade do cliente e do mercado, ou seja, as mudanças de requisitos devem ser refletidas no *software*.

Independentemente de qual modelo de processo esteja sendo utilizado, um conjunto de estruturas de processos genéricos pode ser aplicado à vasta maioria de processos de *software* sem depender da complexidade e tamanho (PRESSMAN, 2006). Esse conjunto de estrutura envolve uma série de atividades denominadas

de atividades guarda-chuva que são aplicadas no decorrer de todo o processo de desenvolvimento de *software* e na forma como as atividades são organizadas e utilizadas, ou seja, sequenciais ou intercaladas: a variação ocorre de acordo com o modelo de desenvolvimento que está sendo utilizado e independe dele. Dessa forma, pode-se dizer que existem, nesse contexto, algumas atividades genéricas. Sommerville (2007) as destaca:

- **Comunicação:** envolve o levantamento de requisitos e outras atividades que necessitam da colaboração dos *stakeholders*;
- **Planejamento:** um plano que descreve tarefas, riscos, recursos necessários, artefatos e cronograma é estabelecido para o desenvolvimento do *software*;
- **Modelagem:** trata de representar o que será construído para que os requisitos e o projeto sejam mais bem entendidos através de modelos;
- **Construção:** trata-se da implementação, a codificação, juntamente com os testes para descobrir as falhas de funcionalidades, comportamento e desempenho do *software*;
- **Implantação:** o *software* construído é entregue ao cliente, que, após avaliação do produto, dá o seu *feedback*.

Dessa forma, vale ressaltar que o desenvolvimento de *software* deve seguir modelos padronizados, independentemente de qual esteja sendo utilizado no projeto. Atividades predefinidas devem ser utilizadas com o objetivo de transformar os requisitos no produto final, de acordo com o escopo também predefinido, a fim de alcançar o produto final com qualidade e no tempo definido.

## Objetivos da metodologia de desenvolvimento e de gestão

Um processo de desenvolvimento de *software* é descrito de forma simplificada por meio de um modelo de processo do mesmo, que também pode ser chamado de ciclo de vida. Esse modelo é basicamente a representação, de forma abstrata, de um processo com o objetivo de explicar as várias abordagens de desenvolvimento (SOMMERVILLE, 2007). O ciclo de vida é documentado e tido como uma metodologia, e baseia-se nas fases do projeto que, por muitas das vezes, são sequências e, em alguns momentos, se sobrepõem, em que o nome e o número devem ser determinados de acordo com a necessidade do gerenciamento, controle

organizacional, natureza do projeto, entre outros fatores (PMBOK, 2008). Como já citado anteriormente, existem vários modelos de desenvolvimento que são utilizados atualmente. Alguns dos mais importantes são: Modelo Cascata, Modelos Incrementais, Modelos Evolucionários (Prototipagem e Espiral) e Desenvolvimento baseado na reutilização.

Para o desenvolvimento de processo, também existe uma variedade de modelos que são utilizados como ciclo de vida para a gestão de melhoria de processos na organização. Essa gestão de processos é uma série de metodologias e técnicas que ajudam as organizações a gerenciarem o seu negócio por meio do conhecimento e compreensão de seus processos. Os modelos de processos de negócio ajudam na formalização de processos, apresentando-os em linguagem comum e em um bom nível de entendimento. Nesse contexto, modelos baseados em *workflow* são bastante utilizados. Não basta apenas implementar um ciclo de vida de *software*; até mesmo para implementá-lo, existe a necessidade de implementar processos que devem ser planejados de forma estruturada, em fases distintas, com recursos e artefatos. Compreender como cada atividade funciona e se relaciona com as demais, seus artefatos e entregas, é de suma importância para garantir a qualidade na entrega do produto final. Alguns dos modelos para projetos de processo mais utilizados por gestores são: PDCA (*plan-do-check-act*), DMAIC (*Define, Measure, Analyse, Improve, Control*), IDEF (*Integrated Computer Aided Manufacturing Definition*) e BPM (*Business Process Management*).

## ATIVIDADE

01. De acordo com o que foi estudado neste capítulo, defina o que são componentes.

02. De acordo com o que foi estudado neste capítulo, conceitue arquitetura de sistemas.

03. De acordo com o que foi estudado neste capítulo, conceitue workflow e processos de software.

## GABARITO

**01.** Componentes são unidades de *software* cujas funcionalidades são responsáveis por executar atividades definidas no sistema. São desenvolvidos e implantados para que atuem de forma independente com a possibilidade de serem combinados entre si, por meio de interfaces, para formar um sistema.

**02.**. A arquitetura de sistemas é a abordagem mais utilizada para representar soluções; dessa forma, se alcança a solução mais adequada ao problema. A arquitetura é baseada em modelo de alto nível para permitir um maior entendimento e uma análise mais fácil do *software* que se pretende desenvolver.

**03.** *Workflow:* representa a automação de processo de negócio, em que documentos, informações ou tarefas são distribuídos de acordo com procedimentos predefinidos. Processos de *software*: formam o alicerce no que diz respeito ao controle gerencial de projetos de software, além de estabelecer o meio para a aplicação de métodos técnicos, a produção de produtos de trabalho, estabelecimento de metas, assegurar a qualidade e a gerência adequada de alterações no projeto.

## REFLEXÃO

Neste capítulo, iniciamos os nossos estudos sobre Arquitetura de Sistemas. Iniciamos o nosso aprendizado com a definição de componentes, em que conhecemos os seus princípios fundamentais e características, além de compreendermos melhor o seu conceito através de dois exemplos que demonstraram bem o que é componente e como eles interagem por meio de suas interfaces. Estudamos também o conceito de arquitetura de sistemas e conhecemos os seus elementos básicos, pelos quais podemos fazer uma ligação entre arquitetura, componentes, conectores e a sua organização. E aprendemos que a arquitetura de sistemas pode ser organizada de diversas formas ou padrões. Conhecemos alguns: estruturas, sistemas adaptáveis, sistemas distribuídos e sistemas interativos. Conhecemos a organização em camadas que faz parte do padrão Estruturas. Com a base adquirida nos estudos sobre componentes e arquitetura de componentes, foi possível compreender de forma mais breve os conceitos de arquitetura de componentes e a sua organização em camadas. Aprendemos as vantagens, os benefícios e os problemas enfrentados por esse tipo de arquitetura. Também estudamos os conceitos dos padrões para o desenvolvimento de modelos de componentes e conhecemos alguns modelos através de exemplos.

Para se aprofundar nos conceitos de Processo de Desenvolvimento, começamos os nossos estudos tratando sobre *Workflow*. Conhecemos as duas definições, conceitos e terminologia básica. Definimos processo de *software* tomando como base os conhecimentos de dois autores renomados na literatura de engenharia de software. Tomamos ciência das atividades comuns aos processos de desenvolvimento e o conceito de atividades guarda-chuva. E, por fim, aprendemos os objetivos de utilizar modelos de processos de desenvolvimento de *software* e modelos de gestão de processos.

Finalizamos este capítulo com uma base de conhecimento solidificado acerca de arquitetura de sistemas e de componentes, além da importância do gerenciamento, seja no ciclo de vida de *software* ou de processos. Com os conhecimentos adquiridos neste capítulo, podemos continuar os nossos estudos neste livro.

## LEITURA

Para você avançar mais o seu nível de aprendizagem envolvendo os conceitos referentes a este capítulo, consulte as sugestões abaixo:


JESUS, V. F. **Arquitetura de Software:** Uma Proposta para a Primeira Aplicação. Londrina: Universidade Estadual de Londrina, 2013.

SEGRINI, B. M. **Definição de Processos Baseada em Componentes**.Vitória: Universidade Federal do Espírito Santo , 2009.

SANTANA, J. W. S. **Sistemas Workflow:** Uma aplicação ao IC. Maceió: Universidade Federal de Alagoas, 2006.

LOURENÇO, F. L. M., BENÍNE, M. A. Estudo do ciclo de vida do software em uma empresa de desenvolvimento de sistemas. In: **Linguagem Acadêmica**. Batatais: Claretiano - Centro Universitário, 2011.

MARTINAZZO, F., KIRCHOFF, D. F., MARTINS, A. R. Q., ROSO, S. C., SPINELLO, S. **O Gerenciamento de Processos de Negócio Aplicado para Melhorar os Resultados Organizacionais** – Um Estudo de Caso Prático. Belém: XXXIV Seminário Nacional de Parques Tecnológicos e Incubadoras de Empresas, 2014.

PIZZA, W. R. **A metodologia Business Process Management (BPM) e sua importância para as organizações**. São Paulo: Faculdade de Tecnologia de São Paulo, 2012.

# REFERÊNCIAS BIBLIOGRÁFICAS


_____. Project Management Coliation (PMI). In: **PMBOK Guide**. **XXXX, YYY, 2008.**

_____. Workflow Management Coliation. In: **The Workflow Reference Model,** Document Number WFMC-TC00-1003 Document status – Issue. Hampshire: Workflow Management Coalition, 1996.

_____. Workflow Management Coliation. In: **The Workflow Management Coliation** – Terminology and Glossary, Document Number WFMC-TC-1011 Document status – Issue. Hampshire: Workflow Management Coalition, 1999.

BACHMANN, F. et al. **Technical Concepts of Component-Based Software Engineering**. Pittsburgh: Carnegie Mellon University, 2000.

BASS, L., CLEMENTS, P., KAZMAN, R. **Software Architecture in Practice**, . Massachusetts: Addison-Wesley, 2003.

BROWN, A. W., WALLNAU, K. C. **Component-Based Software Engineering IEEE Computer Society Press**. Los Alamitos: Wiley-IEEE Computer Society Press, 1996.

CARNEY, D. **Assembling Large Systems from COTS Components:** Opportunities, Cautions, and Complexities. SEI Monographs on the Use of Commercial Software in Government Systems. Pittsburgh: Carnegie Mellon University, 1997.

CHEESMAN, J., DANIELS, J. **UML Components:** a Simple Process for Specifying Component-Based Software. Massachusetts: Addison-Wesley, 2001.

CLEMENTS, P., BACHMANN, F., BASS, L., GARLAN, D., IVERS, J., LITTLE, R., NORD, R., STAFFORD, J. **Documenting Software Architectures.** Pittsburgh:Addison-Wesley, 2004.

DIAS, M.S., VIEIRA, M.E.R. Software architecture analysis based on statechart semantics. In: **International Workshop on Software Specification and Design**. Washington: IEEE Computer Society, 2000.

D'SOUZA, D., WILLS, A. **Objects, Components and Frameworks with UML** – The Catalysis Approach. Boston: Addison-Wesley, 1998.

GARLAN, D. Software architecture: a roadmap. In: **Proceedings of The Conference on The Future of Software Engineering**. Limerick: Carnegie Mellon University, 2000.

GARLAN, D., PERRY, D. Introduction to the Special Issue on Software Architecture. In: **IEEE Transactions on Software Engineering**. Piscataway: IEEE Press, 1995.

GIMENES, I., HUZITA, E. **Desenvolvimento Baseado em Componentes:** Conceitos e Técnicas. Rio de Janeiro: Ciência Moderna, 2005.

HEINEMAN, G. T., COUNCILL, W. T. **Component-Based Software Engineering:** putting the pieces together. Boston: Addison-Wesley, 2001.

KAZMAN, R. Handbook of Software Engineering and Knowledge Engineering. In: **World Scientific Publishing**. Farrer Road: World Scientific Publishing Company, 2001.

KRUCHTEN, P. **Modeling Component with the Unified Modeling Language**. XXXX: International Workshop on Component-Based Software Engineering: **XXX,** 1998.

PRESSMAN, R. S. **Engenharia de Software**. São Paulo: Mcgraw-hill, 2006.

SOMMERVILLE, I. **Engenharia de Software**. São Paulo: Pearson Education-Br, 2007.

SPAGNOLI, L. A., BECKER, K. **Um Estudo Sobre o Desenvolvimento Baseado em Componentes**. Porto Alegre: Pontifícia Universidade Católica - RS, 2003.

SZYPERSKI, C. **Component Software:** Beyond Object-Oriented Programming. Boston, : Addison-Wesley, 2002.

VINCENZI, A. M. R. et al. **Desenvolvimento baseado em componentes:** Conceitos e Técnicas [S.l.]. Rio de Janeiro: Ciência Moderna Ltda., 2005.

WALLNAU, K. et al. On the Relationship of Software Architecture to Software Component Technology. In: **International Workshop on Component-Oriented Programming**. Budapest,: **XXXX,** 2001.

WERNER, C., BRAGA, R. **Desenvolvimento Baseado em componentes**. João Pessoa: , 2000.

# 2

# Definição de requisitos e a aplicação de UML

# Definição de requisitos e a aplicação de UML

## Introdução

É fato que, no decorrer dos anos, a complexidade de sistemas tem aumentado ao mesmo tempo em que houve um aumento no grau de exigência, por parte principalmente dos clientes, acerca da qualidade. Com isso em mente, podemos observar o quão importante é a etapa de definição de requisitos, em que a mesma deve ser realizada com atenção e com o acompanhamento do cliente e dos usuários finais. Tal acompanhamento, o uso de técnicas para a elicitação dos requisitos e para a validação dos mesmos, só tende a beneficiar o nível de qualidade do projeto de *software*. E, para que todos os envolvidos possam compreender o que será desenvolvido e verificar se a necessidade do cliente está sendo atendida, bem como a experiência do usuário final frente ao sistema, a aplicação de UML na arquitetura de sistema é fundamental.

Portanto, neste capítulo aprenderemos sobre a definição de requisitos, os tipos de requisitos, as técnicas para sua elicitação, o processo de validação de requisitos e a utilização da técnica de prototipação, a utilização da UML em arquiteturas de sistemas, modelo conceitual e sobre especificação de interfaces.

## OBJETIVOS

- Aprender o que são requisitos;
- Aprender quais são os tipos de requisitos;
- Conhecer as técnicas de elicitação de requisitos;
- Aprender os principais conceitos que cercam a validação de requisitos;
- Entender os conceitos e a importância da prototipação para a validação de requisitos;
- Entender a UML e a sua importância na arquitetura de sistemas;
- Conhecer um pouco sobre modelo conceitual;
- Compreender a especificação de interfaces.

## Definição de requisitos

Antes de abordarmos os conceitos e a definição de requisitos, é interessante entendermos um pouco sobre a Engenharia de Requisitos. Podemos entendê-la como uma disciplina que compõe a Engenharia de *Software*, tendo como preocupação sistematizar o processo de definição de requisitos. Para Castro (1995), a engenharia de requisitos é uma etapa crucial no que diz respeito a possuir outros conhecimentos além do conhecimento técnico, como, por exemplo, conhecimentos gerenciais, organizacionais, econômicos e sociais. Isso está de certa forma ligado à qualidade do produto (LEE et al., 1998).

Segundo Davis (1993), a engenharia de requisitos tem suas atividades voltadas a produzir uma descrição comportamental externa do sistema que se pretende desenvolver. De acordo com Zave (1997), a engenharia de requisitos está associada à transformação de objetivos do mundo real em regras e funcionalidades em sistemas com o intuito de aumentar a precisão na fase de especificação do comportamento do sistema assim como a sua evolução. Thayer (1997) ressalta a engenharia de requisitos como a primeira etapa do processo de engenharia de *software* e responsável pela coleta, entendimento, armazenamento, verificação e gerência de requisitos.

Para Lamsweerde (2000), a engenharia de requisitos auxilia a identificar quais são as restrições e os objetivos que se pretende alcançar com o sistema que se pretende desenvolver. E Sommerville (2004) a define como processo de descoberta, análise, documentação e verificação de restrições e funcionalidades de sistema de *software*.

Cabe à engenharia de requisitos atuar no processo de desenvolvimento de sistemas de *software* propondo técnicas, métodos e ferramentas que auxiliem na etapa complexa que é a definição de requisitos. Mas, antes de entendermos essa etapa complexa, vamos conceituar o que são requisitos.

Requisitos de sistema de *software*, basicamente, **especificam quais são as funcionalidades** que determinado *software* deve possuir para satisfazer as necessidades de seu cliente, gerando a qualidade do produto. Pfleeger (2004) defende que requisito é uma **característica de um sistema de *software***, ou seja, **uma descrição do que determinado sistema deve ser capaz de fazer para alcançar seus objetivos**. O IEEE possui uma norma, a IEEE Std 610.12-1990 (IEEE, 1990), que traz a seguinte definição de requisitos:

• Uma capacidade ou condição que é necessária para que o usuário resolva determinado problema ou alcance determinado objetivo;

• Uma capacidade ou condição que possua a necessidade de ser atendida ou que deve estar presente no sistema ou componente, e, dessa forma, satisfazer a sua especificação, norma, contrato ou outro documento que pode ser imposto de forma formal;

• Uma representação documentada de determinada capacidade ou condição, de acordo com os itens anteriores.

O interessante dessa definição é que ela aborda duas visões importantíssimas no que envolve os requisitos, que são a visão do usuário e a do desenvolvedor de *software*.

Segundo Jackson (1995), os **requisitos são propriedades do domínio da aplicação, em que essas propriedades são descritas em formatos de diagramas, linguagem natural ou alguma outra forma de notação, desde que isso ajude no entendimento do cliente e do desenvolvedor.**

É durante as etapas iniciais do desenvolvimento de *software* que os requisitos são definidos, sendo eles uma descrição, uma especificação do que deve ser implementado. Ou seja, os **requisitos são responsáveis por descrever como será um comportamento, um atributo, ou uma propriedade do sistema. Além disso, os requisitos podem definir quais são as restrições ao desenvolvimento** (SOMMERVILLE, SAWYER, 1997).

Dessa forma, podemos observar a importância dos requisitos para o desenvolvimento de sistema e que o mesmo deve ser tratado com certo cuidado, pois algum erro aqui pode afetar o sistema negativamente no que diz respeito, principalmente, aos custos e à insatisfação do cliente. Para evitar problemas como esse é que a definição de requisitos deve ser sistematizada, pois a complexidade dos sistemas força e exige que seja dada uma maior atenção ao entendimento do problema, tornando-se o mais eficaz possível. É dever dos engenheiros de *software* entender sobre o local (ambiente) onde o sistema será implantado e escolher os modelos que melhor apresentarão o ambiente (Leite, 1994). Segundo Christel e Kang (1998), a **definição de requisitos é a atividade de maior importância**, ao mesmo tempo **decisiva e crítica** no desenvolvimento de sistema de *software*, principalmente no que envolve a elicitação de tais requisitos.

## Tipos e técnicas de levantamento de requisitos

Existem diversos tipos de requisitos, porém a classificação mais tradicional os divide em três tipos; de acordo com Sommerville (2004), são:

- **Requisitos funcionais:** descrevem as **funções que devem ser oferecidas pelo sistema**, a forma como ele se comportará em determinadas situações. O termo função é utilizado aqui em um sentido amplo da operação que poderá ser realizada, seja através de comandos repassados pelos usuários ou por eventos internos e/ou externos ao sistema. Os requisitos funcionais, em alguns casos, definem de forma explícita o que não deve ser feito pelo sistema. Exemplo: Cadastrar Cliente (CRUD); Gerar Relatório de Vendas;
- **Requisitos não funcionais:** descrevem as **restrições oferecidas pelo sistema**, como restrições de tempo, padrões, processo de desenvolvimento, qualidade, manutenabilidade, desempenho, usabilidade, custos, entre outros. Exemplo: Relatórios gerados no máximo em 10s; Acesso a área logado com certificado digital;
- **Requisitos organizacionais:** estão relacionados de forma direta aos **procedimentos e às políticas organizacionais**, e também com as metas e objetivos organizacionais. Exemplo: Todos os documentos entregues devem seguir o padrão de formatação X.

Esses são os tipos de requisitos mais genéricos e tradicionais; os outros tipos de requisitos podem ser definidos dependendo do projeto e da visão que é necessária para o entendimento dos requisitos, podendo ser aplicáveis ou não. Entre eles, definiremos três tipos:

- **Requisito do usuário:** descrição de uma característica ou comportamento que o **usuário do *software* deseja** que ele possua. Esses requisitos podem ser descritos por um profissional que analisa, entende ou consulta o usuário, ou, ainda, pode ser escrito pelo próprio usuário. Exemplo: Gerar relatórios em linguagem natural com diagramas e tabelas simples;
- **Requisito do sistema:** descrição de uma característica ou comportamento que **é exigido pelo sistema**, levando em consideração o *hardware* e o *software*. Basicamente, é o comportamento esperado do sistema. Geralmente, são definidos por analistas de sistemas ou por engenheiros que fazem o refinamento dos requisitos coletados dos usuários e os transformam em termos de engenharia.

Exemplo: O usuário comum só poderá ter acesso aos relatórios pertencentes ao seu departamento;

- **Requisito do *software*:** descrição de uma característica ou comportamento que **é exigido do *software***. Geralmente, são definidos por analistas de sistemas. Exemplo: Cada pedido deverá conter um único identificador.

A elicitação (ou levantamento) de requisitos é uma das fases da engenharia de requisitos, a primeira. Ou seja, é o início de toda atividade que cerca o desenvolvimento de software. Porém, mesmo sendo a primeira fase, ela não é executada apenas uma vez, mas sim de forma interativa, em que todas as outras etapas da engenharia de requisitos (ver Figura 1) podem possuir o levantamento de requisitos. Para que essa fase seja realizada, existe a necessidade de primeiramente identificar quais serão os usuários finais, ou seja, quem são os usuários que possuem um forte papel para o processo de desenvolvimento de software por meio de suas experiências e conhecimentos (GOGUEN, 1997). Para que esses usuários sejam escolhidos, deve ser levada em consideração a veracidade das informações que podem ser passadas por eles. Essas informações devem ser de confiança, já que isso se trata de informações que serão utilizadas no projeto e devem estar de acordo com a necessidade organizacional. Caso a pessoa errada seja escolhida, isso acarretará em perda de tempo e de dinheiro para ambas as partes. Essa escolha é de extrema importância.

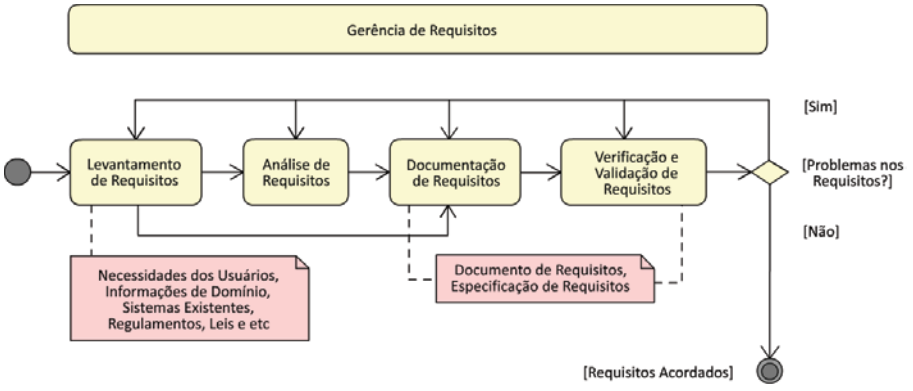


Figura 1. Processo de Engenharia de Requisitos. Adaptado de (Kotonya; Sommerville, 1998).

E os requisitos, ao longo do tempo, podem mudar pois a sua natureza é mutante, já que os clientes tendem a mudar de ideias ao se depararem com outras possibilidades que podem envolver questões sociais, financeiras, psicológicas e/

ou políticas. Além disso, o desenvolvedor precisa conhecer o contexto no qual as informações estão situadas.

Para que a fase de levantamento de requisitos seja eficaz, existe a necessidade de aplicar técnicas para melhorar a comunicação e o entendimento entre os clientes e analistas com o objetivo de que problemas não aconteçam, e, se ocorrerem, que possam ser resolvidos de forma facilitada. Para Jackson (1995), uma **técnica de levantamento de requisitos procura explorar as características de determinada situação-problema e como elas variam**. Como essa fase aborda um contexto social (comunicação), ela não pode ser abordada de forma totalmente tecnológica como na fase que envolve a codificação. Algumas das técnicas de levantamento de requisitos são:

• **Observação:** útil para que novos aspectos do problema possam ser descobertos, tornando-se crucial quando não se possui alguma base sólida para a orientação da coleta de dados. Nessa técnica, existe um **contato direto e prolongado entre o contexto, o ambiente onde o software será inserido, e o observador**. É como se estivesse uma pessoa observando os usuários finais realizando as suas tarefas que poderão ser ajudadas com o novo *software*. E, com isso, essa técnica sofre algumas críticas, pois a inserção de um observador pode modificar a forma como os observados trabalham, além de que o que vai realmente valer é a interpretação pessoal, levando em alguns casos a uma visão distorcida dos fatos ou à parcialidade da realidade por parte do observador;

• **Entrevista:** essa é a técnica de elicitação mais utilizada. É uma técnica eficaz para que seja desenvolvido um **maior e melhor entendimento do problema, permitindo que mais requisitos gerais possam ser coletados**. Aqui os requisitos são discutidos entre os engenheiros ou analistas com diversos usuários para um maior entendimento deles. Kontonya (1998) ressalta dois tipos de entrevistas:

1) **entrevistas fechadas,** em que as questões são predefinidas;

2) **entrevistas abertas,** nas quais não existe uma agenda predefinida, o que permite a discussão dos requisitos de modo aberto. Como nessa técnica os usuários finais relatam seus trabalhos e suas dificuldades, eles acabam adquirindo uma expectativa não realista acerca de como o computador dará suporte a suas tarefas. Desse modo, quando se pretende entender melhor o domínio da aplicação para as questões organizacionais que podem afetar os requisitos, as técnicas de entrevistas são as menos efetivas;

• **JAD:** A técnica *Joint Application Development* é uma marca da IBM que **procura unir dentro de um *workshop* autoridades gerenciais e representativas** para que as decisões possam ser promovidas. JAD é composto em cinco fases: definição do projeto, pesquisa, preparação para a sessão JAD, a sessão JAD e documento final. JAD permite a promoção de cooperação, entendimento e trabalho em grupo entre os muitos grupos de usuários e os analistas;

• **PD:** A técnica *Participatory Design* permite a **inclusão de fatores técnicos-sociais**, em que o projeto é desenvolvido com o usuário. Os trabalhadores e os clientes contribuem de forma produtiva para as organizações, sendo encorajados a expressar seus desejos e a exercitar suas capacidades de tomada de decisão, assumindo a responsabilidade do impacto de suas ações;

• **QFD:** A técnica *Quality Function Deployment* aborda um conceito que **permite a interpretação dos requisitos do cliente em requisitos técnicos**. As fases iniciais dessa técnica podem ser descritas como sistema de identificação e priorização das necessidades de acordo com a avaliação de cada recurso. O conceito de QFD é aplicado no levantamento de requisitos, em que o cliente é o guia para a criação de requisitos;

• **CRC:** A técnica *Cooperative Requirements Capture*, assim como o JAD, é uma sessão de grupo que possui papéis claramente definidos. Nessa técnica, os participantes são os usuários, o facilitador e outros envolvidos de forma indireta no sistema. O CRC diferencia-se do JAD e QFD por **focar no usuário operativo**;

• **Prototipação:** Essa técnica utiliza-se de protótipos que levam em consideração um conjunto inicial de requisitos para que sejam avaliados pelo usuário. Esse conjunto de **requisitos são traduzidos em um protótipo de interface do usuário**, que é criado e mantido pelo projetista da forma mais simples possível. A maior vantagem dessa técnica é que ela permite apresentar uma variedade de alternativas antes que se gaste esforço com a implementação/codificação de algum protótipo. E somente após os usuários finais aceitarem o protótipo é que será criado o documento de especificação dos requisitos;

• **Cenários:** Para alguns, a utilização dessa técnica facilita por se relacionar mais a exemplos reais do que a descrições de funcionalidades do sistema. Por isso, é bastante útil criar uma variedade de interação dos cenários e utilizá-los para levantar e compreender melhor os requisitos de sistema. Os **cenários exemplificam as sessões de interação entre os usuários finais e o sistema**, permitindo que eles simulem suas interações por meio dos cenários. Dessa forma, os usuários explicam

o que estão fazendo e a informação que precisam que o sistema dê para possam descrever a tarefa descrita no cenário.

Tabela 1. Resumo de algumas das técnicas de levantamento de requisitos.

| TÉCNICA | DESCRIÇÃO |
|---|---|
| OBSERVAÇÃO | Nessa técnica, existe um contato direto e prolongado entre o contexto, o ambiente onde o *software* será inserido, e o observador. É como se estivesse uma pessoa observando os usuários finais realizando as suas tarefas que poderão ser ajudadas com o novo *software*. |
| ENTREVISTA | É a técnica de elicitação mais utilizada. É uma técnica eficaz para que seja desenvolvido um maior e melhor entendimento do problema, permitindo que mais requisitos gerais possam ser coletados. |
| JAD | Permite a inclusão de fatores técnicos-sociais, em que o projeto é desenvolvido com o usuário. Os trabalhadores e os clientes contribuem de forma produtiva para as organizações. |
| QFD | Permite a interpretação dos requisitos do cliente em requisitos técnicos. As fases iniciais dessa técnica podem ser descritas como sistema de identificação e priorização das necessidades de acordo com a avaliação de cada recurso. |
| CRC | Seção de grupo que possui papéis claramente definidos. Nessa técnica, os participantes são os usuários, o facilitador e outros envolvidos de forma indireta no sistema. |
| PROTOTIPAÇÃO | Leva em consideração um conjunto inicial de requisitos para que sejam avaliados pelo usuário. Esse conjunto de requisitos é traduzido em um protótipo de interface do usuário. |
| CENÁRIOS | Exemplifica as sessões de interação entre os usuários finais e o sistema, permitindo que eles simulem suas interações por meio dos cenários. |

## Validação de requisitos e prototipação

A Validação de Requisitos é a última atividade da engenharia de requisitos e **objetiva mostrar os requisitos definidos para o sistema**. A validação de requisitos é uma etapa fundamental que deve ser realizada com maior ênfase na sua completude e consistência. As mudanças e alterações podem significar um custo consideravelmente alto, principalmente se ocorrerem ao final de todo o processo de desenvolvimento. Portanto, nesse processo os possíveis problemas de requisitos devem ser identificados, sendo de suma importância a identificação antes da produção do documento que servirá como base do desenvolvimento do sistema.

Segundo Kontonya (1998), as checagens de requisitos devem ser realizadas tendo cuidado com o objetivo de garantir a integralidade e consistência dos mesmos. **A validação deve propiciar o consenso entre os *stakeholders***, pois é provável que existirão objetivos conflitantes. Ao validar os requisitos, se confirmará a veracidade presente no documento de requisitos, e isso é crucial: essa validação deve também alcançar suas especificações complementares. Todos os requisitos e suas documentações devem refletir a real necessidade dos usuários finais e o cliente, sendo autenticadas, pois serão a base para as próximas fases do processo de desenvolvimento de *software*.

Loucopoulus (1995) define esse processo como **atividade certificadora do documento de requisitos**, afirmando que o mesmo está de acordo com as necessidades dos seus usuários. A descrição dos requisitos deve estar de forma clara e explícita não apenas para que seja validado, mas também para solucionar possíveis conflitos que possam existir. Para Sommerville (2004), é de suma importância a existência da validação ao identificar as funcionalidades que são exigidas para o atendimento dos vários usuários; consistência com o intuito de verificar a existência de requisitos que possam ter descrições diferentes para uma mesma funcionalidade no sistema; inclusão completa de todas as restrições e funcionalidades que foram solicitadas pelo cliente; a implementação deve estar assegurada, bem como a facilidade da verificação, pois ela pode minimizar as divergências que possam existir entre o fornecedor e o cliente. Lamsweerde (1998) aponta a complexidade da validação de requisitos por essa ser uma tarefa que abrange mais o social e pela sua relação difícil em alcançar consenso entre os objetivos conflitantes dos diferentes usuários.

Nesse ponto, podemos afirmar que, **para a validação de requisitos, é necessário que ela seja realizada em vários momentos ou sessões de trabalho**.

É como se fosse um processo de lapidação para encontrar os pontos concordantes acerca dos requisitos e visualizar as implicações de decisões futuras. Dessa forma, é essencial a participação de especialistas para orientar os clientes, usuários e desenvolvedores a fim de resolver seus conflitos. E isso deve ser feito em vários momentos, o que torna esse processo contínuo, tendo um maior esforço nas fases de levantamento e especificação de requisitos. Como também é um processo interativo, é aplicado tanto no modelo final de requisitos quanto nos modelos ditos intermediários durante todo o processo de requisitos.

A validação de requisitos **deve gerar um documento bem definido para que as incoerências e inconformidades sejam corrigidas** a fim de que se possa dar continuidade ao processo de desenvolvimento; assim, a validação reduz o tempo que seria gasto para identificar essas incoerências e inconformidades. **A validação possui um alto grau de eficiência na descoberta de erros e também reduz a possibilidade de encontrá-los em alguma fase final**, o que poderia gerar algum desastre no trabalho. É desafiante para esse processo ter de mostrar a especificação correta do sistema, pois ele não possui uma representação que possa ser utilizada como base da mesma forma que o projeto e a implementação possuem o documento de requisitos. Ou seja, formalmente, não há uma maneira para demonstrar que a especificação está de fato correta (KONTONYA, 1998).

Podemos citar alguns problemas que somente são identificados durante o processo de validação de requisitos:

- A não conformidade com padrões de qualidade;
- Requisitos incertos, ambíguos;
- Erros em modelos do problema ou sistema;
- Conflitos entre os requisitos que não foram identificados durante a análise.

Todos esses problemas, obrigatoriamente, devem ser resolvidos antes que o documento de requisitos seja aprovado e utilizado para o desenvolvimento do produto de *software*. Em alguns casos, os problemas são corrigidos para que seja melhorada a descrição dos requisitos. Já em outros, os problemas foram gerados de equívocos durante o levantamento, análise e modelagem de requisitos (KOTONYA, 1998).

Embora para a validação não exista algum documento que possa ser utilizado como base, existe uma forma demonstrar a conformidade da especificação de requisitos. A figura a seguir ilustra o processo de validação de requisitos.


Figura 2. Entradas e Saídas do Processo de Validação de Requisitos.
Adaptado de (Kotonya, 1998).

Assim como para a elicitação de requisitos, a validação possui uma variedade de técnicas que, ao serem aplicadas, dão todo um suporte para esse processo. Daremos ênfase à técnica de Prototipação, que nada mais é do que a simulação de telas para que o usuário possa visualizar como, teoricamente, será a aplicação que ainda não foi produzida. Observe que também temos essa técnica no levantamento de requisitos; portanto, caso essa técnica tenha sido utilizada na elicitação, faz sentido utilizá-la para validar os requisitos. Mas, dessa vez, na validação, os protótipos estarão mais robustos e completos, devendo conter todos os requisitos necessários para garantir que esteja tudo de acordo com o esperado pelos usuários finais e cliente. No levantamento de requisitos o protótipo pode apresentar a ausência de algumas funcionalidades e ainda não possuir as alterações solicitadas ao longo do processo de análise de requisitos. Portanto, existe a necessidade de que esse protótipo tenha o seu desenvolvimento continuado durante a validação dos requisitos (KOTONYA, 1998).

Os protótipos auxiliam bastante os usuários por eles conseguirem identificar de forma mais clara e exata o que eles querem, aproximando-se assim do produto final. Desse modo, **por meio da utilização de protótipos, fica mais fácil identificar as incoerências, inconformidades e outros problemas que os**

**requisitos possam apresentar**. Além disso, essa técnica melhora a comunicação entre os desenvolvedores e os usuários.

Porém, a técnica de prototipação na validação de requisitos apresenta algumas desvantagens:

• É possível que a sua implementação traga desilusões aos usuários finais, principalmente se as interfaces da versão final não estiverem de acordo com as utilizadas no protótipo;

• Implementar protótipo demanda esforço e tempo, ou seja, deve ser verificado se todo o esforço e tempo gasto serão compensados pelas vantagens do protótipo;

• Como o protótipo aumenta a comunicação entre os desenvolvedores e os usuários, isso pode gerar uma mudança de foco: ele estará mais voltado para a interface do que para o fato de o sistema estar realmente atendendo o que o processo de negócio necessita.

Na figura a seguir, podemos visualizar a técnica de prototipação no processo de validação de requisitos; em seguida, temos a descrição de suas atividades:

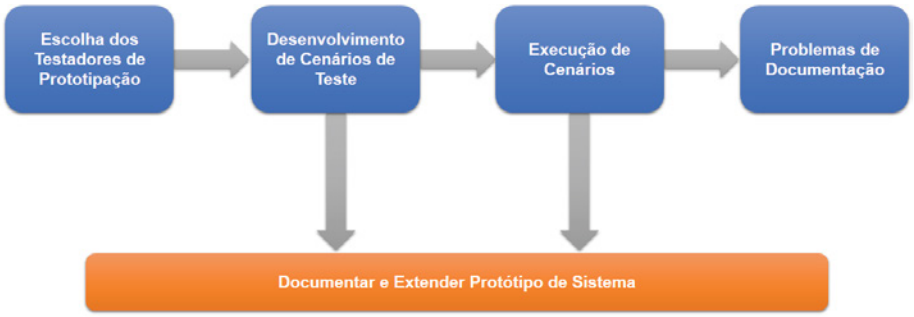


Figura 3. Validação de Requisitos por Prototipação. Adaptado de (KOTONYA, 1998).

• **Escolha dos testadores de prototipação:** a equipe de testes é escolhida, e os melhores testadores são aqueles que não possuem familiaridade com o sistema;

• **Desenvolvimento de cenários de teste:** os cenários elaborados devem cobrir todas as situações possíveis tendo como ponto de partida os requisitos;

• **Execução de cenários:** os cenários são executados por conta própria pelos usuários do sistema. Dessa forma, se obtém a melhor perspectiva, pois assim tem-se uma situação mais realista;

• **Problemas de documentação:** os problemas encontrados devem ser documentados para que sejam analisados; essa coleta de problemas deve ser realizada de forma cautelosa.

## UML na arquitetura de sistemas

Várias metodologias de desenvolvimento de sistemas surgiram, desde 1980, com o intuito de auxiliar a orientação a objetos. Os pesquisadores James Rumbaugh, Ivar Jacobson e Grady Booch, no começo dos anos 90, reuniram características presentes em suas técnicas de modelagem, o que resultou em um padrão de referência para a modelagem orientada a objetos chamado de UML (*Unified Modeling Language* – Linguagem de Modelagem Unificada).

Essa linguagem é basicamente a fusão dos métodos OMT (*Object Modeling Technique*), de Booch e Rumbaugh, e OOSE (*Object-Oriented Software Engineering*), de Jacobson. Essa união baseou-se nos pontos fortes dessas técnicas, iniciando-se com Booch e Rumbaugh, que lançaram em 1995 o UM (*Unified Method*); em 1996, a UML 0.9 foi lançada já com a participação de Jacobson. No entanto, a UML apenas tornou-se um padrão mundial em 1997 com a aprovação da Agência Americana de Padrões – *Object Management Group* (OMG).

Conceitualmente, segundo Booch, Jacobson e Rumbaugh (2000), a UML trata-se de **uma linguagem padronizada voltada para elaborar a estrutura de projetos de *software*** (ou seja, a arquitetura de sistema de *software*). Com o que aprendemos no Capítulo 1 deste livro sobre arquitetura de sistemas, já podemos inferir que a UML é utilizada para visualizar, especificar, construir e documentar os artefatos que fazem parte de sistemas de *software*. Basicamente, a UML é uma linguagem **utilizada para descrever por meio de modelos uma arquitetura de *software*** de forma simplificada, realista, apresentando uma perspectiva bem específica para que proporcione uma melhor compreensão acerca do sistema. Ela vai ser **responsável por facilitar visualmente o entendimento de todos acerca do projeto**. São vários os diagramas que fazem parte da UML, e cada um deles possui o seu diferente nível de precisão responsável por uma descrição e representação de parte do sistema.

A UML abrange a representação gráfica com diagramas por meio de técnicas de modelagem que utilizam elementos (objetos, classes, atributos etc.) presentes na orientação a objetos. A UML não é apenas um modelo de engenharia de *software* e muito menos é exclusivo a uma etapa do processo de desenvolvimento como na etapa de engenharia de requisitos. Ela se apoia no desenvolvimento incremental, e esses modelos evoluem com a inserção de novos detalhes.

As principais características da UML, de acordo com Booch, Jacobson e Rumbaugh (2000), são:

- **Centrado na arquitetura:** a arquitetura de sistemas é o artefato-chave para conceituar, construir, gerenciar e evoluir um sistema em desenvolvimento, que, ao mesmo tempo, representa uma visão completa do projeto. A UML leva em consideração todos os fatores que influenciam na arquitetura de sistemas, como, por exemplo, plataforma de software (SO, SGBD, protocolos de comunicação em rede etc.), blocos de construção reutilizáveis (componentes e frameworks), requisitos não funcionais, fatores de distribuição e sistemas legados;
- **Orientado a casos de uso:** o diagrama de casos de uso é o artefato principal e responsável para determinar de modo preciso o comportamento que se deseja para o sistema;
- **Processo Iterativo:** a UML beneficia o gerenciamento de versões que são executáveis e incrementais, em que cada versão inclui as melhorias incrementais. Cada interação completada resulta na liberação de uma nova versão do sistema.

Quatorze são as técnicas de modelagem que a UML 2.0 envolve. Essas técnicas são divididas em dois grupos: técnicas de modelagem estruturais (ressaltam a estrutura de elementos) e técnicas de modelagem comportamentais (ressaltam o comportamento e interação entre elementos). Na tabela a seguir, podemos conhecer a divisão entre esses dois grupos.

Tabela 2. Diagramas UML e a sua divisão entre técnicas de modelagem estruturais e comportamentais.

| TÉCNICA DE MODELAGEM | DIAGRAMAS |
|---|---|
| DIAGRAMAS DE ESTRUTURA (ESTÁTICOS) | Diagrama de Classe |
| | Diagrama de Objeto |
| | Diagrama de Pacotes |
| | Diagrama de Componentes |
| | Diagrama de Implantação |
| | Diagrama de Estrutura Composta |
| | Diagrama de Perfis |

| DIAGRAMAS DE COMPORTAMENTO (DINÂMICOS) | Diagrama de Casos de Uso |
|---|---|
| | Diagrama de Atividade |
| | Diagrama de Máquina de Estados |
| | Diagrama de Interação (Diagrama de Sequência) |
| | Diagrama de Interação (Diagrama de Comunicação) |
| | Diagrama de Interação (Diagrama de Tempo) |
| | Diagrama de Interação (Diagrama de Interação Geral) |

## Exemplo de uso da UML

O Diagrama de Casos de Uso é composto por casos de usos (representados por elipses), atores (representados por bonequinhos), o sistema (representado por um retângulo) e as interações (linhas e setas) entre esses elementos. A sua utilização está direcionada para a descrição de um conjunto de cenários, a captura de requisitos e a delimitação do escopo do sistema. A figura a seguir ilustra um exemplo de diagrama de casos de uso de um sistema de vendas.

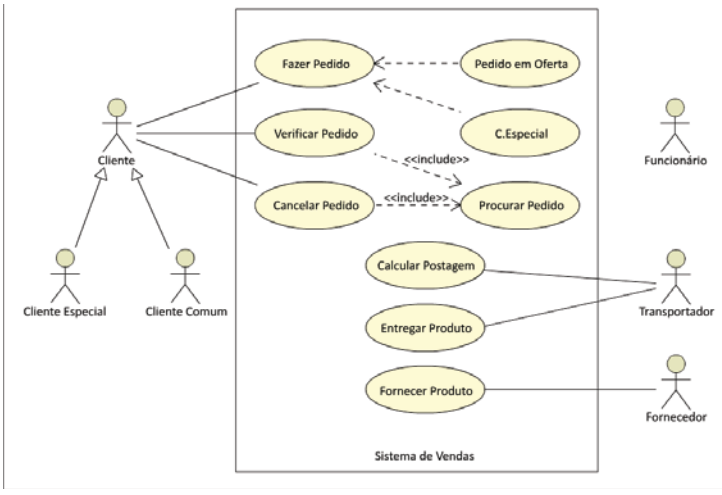


Figura 4. Diagrama de casos de uso de um sistema de vendas.

Logicamente, antes da criação de um diagrama UML, os requisitos já devem ter sido coletados. Esse tipo de diagrama pode ser utilizado na técnica de cenários da elicitação de requisitos para capturar novos requisitos; por fim, ao atingir a validação dos requisitos, ele será responsável por delimitar o escopo do projeto, ou seja, representará o que o sistema deverá fazer, e o que não estiver nele poderá ser considerado como o que o sistema não precisa fazer. Observando a figura acima, temos um diagrama representando um sistema de vendas com possui cinco atores, sendo o Cliente Especial e o Cliente Comum tipos do ator Cliente, além de nove casos de uso (Fazer Pedido, Verificar Pedido, Cancelar Pedido, Pedidos em Oferta, Cliente Especial, Procurar Pedido, Calcular Postagem, Entregar Produto, Fornecer Produto) que representam as funcionalidades do sistema. E, por fim, temos as interações entre casos de usos e atores, em que a esses últimos são delegados suas responsabilidades diante das funcionalidades do sistema. Por exemplo, o ator Funcionário é o único que não vai interagir diretamente com o sistema, enquanto o autor Cliente (Cliente Especial e Cliente Comum) poderá fazer pedido (podendo opcionalmente fazer pedido em oferta ou como cliente especial), verificar pedido e cancelar pedido (em ambos, obrigatoriamente terá de procurar o pedido). Já o ator Transportador será responsável por calcular a postagem e entregar o produto, e, por último, o ator Fornecedor será responsável por fornecer o produto.

Com isso, podemos observar que um único diagrama pode nos dizer muito. Claro que no documento de projeto arquitetural os diagramas são descritos de uma melhor forma. Mas observe que a visualização gráfica dos requisitos torna o entendimento das funcionalidades, interações e escopo do projeto mais fácil para a compreensão do que uma tabela de requisitos repleta de textos. Embora existam quatorze diagramas UML, não necessariamente serão criados quatorze diagramas em um projeto arquitetural: a quantidade de diagramas UML que precisarão ser construídos depende da complexidade e das visões do sistema que será desenvolvido. Em alguns projetos, por exemplo, em relação ao diagrama de casos de uso, alguns casos de uso que compõem um diagrama deverão ser detalhados em outros diagramas de casos de uso exclusivamente para eles.

Outro exemplo de aplicação de UML está presente na Figura 1 deste capítulo, em que consta a ilustração do processo de engenharia de requisitos. Trata-se do diagrama de atividades.

## Modelo conceitual

Modelo conceitual integra as atividades que estão relacionadas com o projeto de interação entre o usuário e a interface. Ele é baseado nos requisitos e tem como objetivo a criação de um sistema de acordo com os objetos, propriedades e relações mapeadas que abrangem as tarefas dos usuários. Ele **deve ser estabelecido para que seja compreendido pelo usuário da forma com que se pretende projetar**. Um modelo do mundo real deve fazer parte do produto da interface, ou seja, **são utilizados cenários, metáforas que simulam e estabelecem alguma relação com a utilização em determinada tarefa do usuário final**. O modelo conceitual possui suporte dos paradigmas de interação para definir qual é a tendência de utilização do produto que está sendo desenvolvido, no caso, a interface.

O modelo conceitual é formado por diversas suposições extraídas no mundo real que apontam quais são as regras de negócio de determinado sistema. Nessa etapa, os protótipos e a boa escolha de tecnologias devem estar de acordo para o entendimento dos processos. Basicamente, trata-se de descrever um sistema respondendo a algumas perguntas, como: o que sistema deve fazer? Como ele deve se comportar? Como ele deve parecer?

Esse tipo de modelo auxilia a mostrar quais são as conexões-chave nos processos e sistemas para que sejam inseridas características ainda mais específicas a partir do desenvolvimento de modelos com maior grau de complexidade. **Desenvolver um modelo conceitual é somar o que foi entendido do produto com as necessidades do usuário e somar os outros requisitos que foram identificados através de suposições**. A qualidade de uma interface está diretamente ligada ao seu modelo conceitual, pois o que importa é a experiência do usuário frente ao produto de interface, não importando o tamanho ou tipo de código. Para que o usuário possa ter uma boa experiência e assim alcançar uma boa qualidade de interface, ele possui o papel fundamental para isso, pois deve-se entender as rotinas, manias e o contexto do usuário, além de seu comportamento frente a interface; dessa forma, é possível desenvolver e/ou definir o produto dentro de um contexto fácil de ser reconhecido e as tarefas possíveis de executar. Para desenvolver o modelo conceitual, primeiramente devemos compreender o usuário e procurar as melhores soluções para o projeto.

## Especificação de interfaces

Ao especificar interfaces, é interessante tratá-las separadamente da lógica de programação. Dessa forma, trata-se apenas da representação relacionada à interação entre a aplicação e o usuário final longe dos detalhes da codificação. Ou seja, a **definição de interface de forma independente do ambiente e linguagem de programação.**

A interface deve ser descrita em um nível abstrato e gerada para ser interpretada em diferentes telas de dispositivos, como PDAs, PCs, notebooks, smartphones etc. Conceitualmente, existem dois tipos de abordagens para especificações de interfaces que serão tratados aqui: abordagens associadas a métodos de projeto e abordagens não associadas a métodos de projeto. O primeiro reúne as abordagens que são baseadas em modelos para o desenvolvimento de interface de usuário. Entre essas abordagens, destacamos a Teresa XML (*eXtensible Markup Language*), o XIML (*eXtensible Interface Markup Language*) e, por último, o WebML (*Web Markup Language*).

A Teresa XML (Berti, et al., 2004) trata-se de uma ferramenta que possui um **ambiente semiautomático e completo voltado à modelagem e à geração de interfaces de aplicações tomando como base a linguagem XML**. A Teresa XML possui três níveis de abstração voltados à geração de interfaces: modelo de tarefa, modelo de interface abstrato e modelo de interface concreta. Existe a definição de linguagem específica para cada nível citado. Porém, o nível com menor grau de abstração (interface concreta) evidencia a desvantagem dessa ferramenta por ela ser dependente de plataforma. Sendo assim, outras variáveis devem ser levadas em consideração ao se desenvolver interface de acordo com a plataforma.

O XIML (Abrams, 1999) também se trata de uma representação que possui o XML como base. Essa linguagem **tem como visão o suporte universal para funcionalidade de aplicações durante todo o ciclo de vida de determinada interface de usuário**. A sua principal meta é o desenvolvimento fundamentado em modelos. Como a linguagem na qual é baseada, o XIML disponibiliza a troca de dados entre aplicações a partir do projeto até a operação e manutenção de interfaces. Por permitir esse intercâmbio de dados, o XIML permite a transformação de modelos em soluções multiplataformas da implementação. Ao descrever uma interface com essa linguagem, obtém-se a composição de aspectos a nível abstrato e a um nível mais concreto acerca de detalhes da codificação da interface. Assim como a Teresa XML, o XIML possui três modelos em sua

definição para o desenvolvimento de interfaces: modelo de tarefa, modelo de plataforma e, por último, modelo de apresentação.

Assim como as demais, as especificações de interface do WebML (WebRatio Team, 2002) possuem base na linguagem XML e são independentes de plataforma e linguagem de programação. Os elementos são representados em XML para a codificação e manutenção de sites de forma automática. Combinando com a tecnologia XSL, ao se especificar com WebML **existe a possibilidade de transformação em templates de páginas**, sendo eles transcritos para determinada linguagem escolhida de acordo com o seu suporte. Elementos visuais são utilizados pelo WebML com o intuito de representar informações em uma página, na qual tais informações foram descritas no modelo de dados da linguagem, podendo representar conteúdo ou operações. As interfaces que fazem uso dessa linguagem são multiplataformas, independentemente de plataforma e linguagem de programação, e o XML, de forma abstrata, faz descrição que abrange todos os elementos e operações.

Já a segunda abordagem que trata das não associadas a métodos de projetos possui fundamento em modelos voltados à criação de interfaces e geralmente possui um nível de menor grau quando se trata de descrever interfaces. Entre as linguagens que possuem mais destaque, trataremos de: XUL (*XML User Interface*), UIML (*User Interface Markup Language*), XAML (*eXtensible Applications Markup Language*) e o PIMA (*Platform Independent Model for Applications*) da IBM.

Com o XUL (Harjono, 2001), tem-se a **possibilidade de construir facilmente aplicações Web sofisticadas, ricas, de fácil implementação e manutenção**, por serem baseadas em XML. Com menor esforço, as especificações de interfaces definidas com XUL podem ser transformadas em outras linguagens. E, assim como o HTML, essa ferramenta permite a utilização de CSS *Style Sheet* e *JavaScript*, além de XSLT, Xpath e o DOM.

A UIML (Ali et al., 2002) é uma linguagem declarativa para a descrição de interface e também baseada em XML. O seu objetivo é a **criação de interfaces multiplataformas permitindo a codificação de interfaces sem a necessidade de aprender linguagens e APIs** (Aplicações de Programação de Interfaces) específicas. A UIML separa o código lógico da aplicação do código da interface. Essa linguagem não descreve interfaces de maneira tão abstrata.

A XAML (Dalal, 2011) permite a **definição de interfaces com sintaxe XML**, tratando a lógica de programação do sistema de forma separada da interface, em que é possível a utilização de *JavaScript* e CSS. Assim como o XUL, o seu nível de definição é pouco abstrato. A XML depende da plataforma da *Microsoft Windows*.

O Projeto PIMA, da IBM, **procura especificar aplicações sem depender de plataforma**, ou seja, plataforma independente. E, para o desenvolvimento de determinada aplicação com o PIMA, tal aplicação deverá conter um ciclo de vida que possui três partes: *design-time, load-time e run-time*. O *design-time* é a fase de identificação dos elementos abstratos que interagem, sendo responsável por modelar e desenvolver o sistema. A *load-time* compõe o sistema, adapta e carrega em dispositivos os componentes que pertencem ao sistema. E o *run-time* é a execução do sistema por parte do usuário, em que um dispositivo inicia a utilização de funções do sistema.

## ATIVIDADE

**01.** De acordo com o que foi estudado neste capítulo, conceitue definição de requisitos.

**02.** De acordo com este capítulo, quais são os tipos e técnicas de levantamento de requisitos?

**03.** De acordo com o que foi estudado, neste capítulo, qual a importância da validação dos requisitos?

**04.** De acordo com o texto, qual a importância da UML na arquitetura de sistemas?

**05.** De acordo com o texto, o que é modelo conceitual e quais os tipos de abordagens para especificações de interfaces?

## GABARITO

**01.** Uma descrição, uma especificação do que deve ser implementado. Ou seja, os requisitos são responsáveis por descrever como será um comportamento, um atributo ou uma propriedade do sistema. E os requisitos também podem definir quais são as restrições ao desenvolvimento.

**02.** Tipos de requisitos: requisitos funcionais, requisitos não funcionais, requisitos organizacionais, requisitos dos usuários, requisitos do sistema e requisitos do software.

Técnicas de levantamento de requisitos: observação, entrevista, JAD, PD, QFD, CRC, cenários e prototipação.

**03.** Importância de atividade certificadora do documento de requisitos que afirma que o mesmo está de acordo com as necessidades dos seus usuários. É de suma importância para identificar as funcionalidades exigidas para o atendimento dos vários usuários. A validação possui um alto grau de eficiência na descoberta de erros e também reduz a possibilidade de encontrá-los em alguma fase final, o que poderia gerar algum desastre no trabalho.

**04.** A UML é uma linguagem utilizada para descrever por meio de modelos uma arquitetura de software de forma simplificada, realista, apresentando uma perspectiva bem específica para que proporcione uma melhor compreensão acerca do sistema. Ela vai ser responsável por facilitar visualmente o entendimento de todos acerca do projeto.

**05.** Modelo conceitual: é um modelo baseado nos requisitos e tem como objetivo a criação de um sistema de acordo com os objetos, propriedades e relações mapeadas que abrangem as tarefas dos usuários. Um modelo conceitual é composto por diversas suposições extraídas no mundo real que apontam quais são as regras de negócio de determinado sistema: trata-se de descrever um sistema respondendo a algumas perguntas, como: o que sistema deve fazer? Como ele deve se comportar? Como ele deve parecer?

Tipos de especificação de interfaces: abordagens associadas a métodos de projeto e abordagens não associadas a métodos de projeto.

## REFLEXÃO

Neste capítulo, iniciamos os nossos estudos sobre Definição de Requisitos e Aplicação de UML. Iniciamos o nosso aprendizado com a definição de requisitos, quando conhecemos um pouco sobre engenharia de requisitos, além de conceitos presentes na literatura sobre ambos. Conhecemos também os tipos de requisitos, a sua importância e algumas técnicas da fase de levantamento ou elicitação de requisitos. E aprendemos quão importante é validar os requisitos junto ao cliente. Entre algumas técnicas de validação dos requisitos, o foco de nossos estudos foi a técnica de prototipação: conhecemos as suas vantagens e desvantagens, além de visualizarmos a técnica de prototipação no processo de validação de requisitos.

Compreendemos também a importância de se utilizar representação gráfica, seja para levantar requisitos os validá-los, e, sobretudo, a sua importância no projeto arquitetural do sistema. Nesse ponto dos estudos, podemos fazer um link com o que foi estudado no capítulo anterior, em que se tratou do uso de modelos para o reconhecimento da abstração, quando todos os envolvidos no projeto poderiam obter de forma facilitada o entendimento acerca do que será desenvolvido. Mostramos que a UML é uma aliada para a arquitetura de sistemas de software. Conhecemos suas características e verificamos que existem quatorze representações gráficas ou diagramas UML divididos em duas técnicas de modelagem. Aprendemos um deles, o diagrama de casos de uso, em um exemplo prático de sistemas de vendas.

Ao partir para Modelo Conceitual, aprendemos que esses modelos são baseados nos requisitos e procuram criar sistemas de acordo com os objetos, propriedades e relações mapeadas que abrangem as tarefas dos usuários. Um modelo do mundo real deve fazer parte do produto da interface, ou seja, nele são utilizados cenários e metáforas que simulam e estabelecem alguma relação com a utilização em determinada tarefa do usuário final. Aprendemos também que, para especificar interface, é interessante trabalharmos apenas com ela independentemente da lógica de programação das regras de negócio. Deve-se trabalhar com a especificação de interface de forma "isolada" do resto do sistema. E conhecemos algumas ferramentas que fazem muito bem isso, procurando ser multiplataforma e atender a essa especificação com qualidade. Algo interessante que podemos notar é o uso da tecnologia XML para essa etapa, quando se utiliza o seu potencial de interoperabilidade e independência de linguagens de programação e plataforma.

Finalizamos este capítulo com um entendimento melhor sobre requisitos, seus tipos e técnicas de levantamento, validação de requisitos e a utilização da técnica de prototipação, a utilização da UML em arquiteturas de sistemas, modelo conceitual, além da especificação

de interfaces e ferramentas para o seu suporte. Com os conhecimentos adquiridos neste capítulo, podemos dar continuidade aos nossos estudos neste livro.

## LEITURA

Para você melhorar o seu nível de aprendizagem envolvendo os conceitos referentes a este capítulo, consulte as sugestões abaixo:

JUNIOR, D. P. A., CAMPOS, R. **Definição de requisitos de software baseada numa arquitetura de modelagem de negócios.** Disponível em: http://hdl.handle.net/11449/29171. Acesso em: XXXXX.

ESPINDOLA, R. S., MAJDENBAUM, A., AUDY, J. L. N. Uma Análise Crítica dos Desafios para Engenharia de Requisitos em Manutenção de Software. In: **VII Workshop on Requirements Engineering** Tandil: Grupo Editor Tercer Milenio S.A., 2004.

LOPEZ, M. R. F. **Estudo da Prototipação na Engenharia de Requisitos para Desenvolvimento de softwares Interativos em Ciclo de Vida Acelerado.** São Paulo: Estudo da Prototipação na Engenharia de Requisitos para Desenvolvimento de softwares Interativos em Ciclo de Vida Acelerado, 2003.

BOOCH, G., RUMBAUGH, J., JACOBSON, I. **UML:** Guia do Usuário. Rio de Janeiro: Elsevier, 2006.

FILHO, J. L., IOCHPE, C. Um estudo sobre modelos conceituais de dados para projeto de bancos de dados geográficos. In: **Revista IP-Informática Pública, v. 1, n. 2.** Belo Horizonte: Empresa de Informática e Informação do Município de Belo Horizonte, 1999.

LEITE, J. C., DE SOUZA, C. S. Uma linguagem de especificação para a engenharia semiótica de interfaces de usuário. In: **Workshop Sobre Fatores Humanos em Sistemas Computacionais**. XXXX: YYYY, 1999.

## REFERÊNCIAS BIBLIOGRÁFICAS

_____. IEEE, IEEE Std 610.12. In: **IEEE Standard glossary of software engineering terminology**. New York: IEEE Computer Society, 1990.

_____. WebRatio Team. In: **WebML User Guide**. Tutorial fornecido pela ferramenta WebRatio 4.2. XXX: YYY, 2002.

ABRAMS, M. et al. Appliance-Independent XML User Interface Language. In: **Proceedings of the Eighth International World Wide Web Conference**. Toronto: XXXX, 1999.

ALI, M. F., QUIÑONES, M. A. P., ABRAMS, M., SHELL, E. Building Multi-Platform User Interfaces with UIML. In: **Computer-Aided Design of User Interfaces III**. Dordrecht: Kluwer Academic Publishers, 2002.

BERTI, S., CORREANI, F., PATERNÒ, F., SANTORO, C. The TERESA XML Language for the Description of Interactive Systems at Multiple Abstraction Levels. In: **Proceeding Workshop on Developing User Interfaces with XML:** Advances on User Interface Description Languages. Gallipoli: ACM, 2004.

BOOCH, G., RUMBAUGH, J., JACOBSON, I. **UML, Guia do Usuário.** Rio de Janeiro: Campus ,2000.

CASTRO, J. F. B. Introdução à engenharia de requisitos. In: XXI Congresso da Sociedade Brasileira de Computação. Canela: XXX, 1995.

CHRISTEL, M., KANG, K. **Issues in Requirements Elicitation.** Pittsburgh: Software Engineering Institute, 1998.

DALAL, M., GHODA, A. **XAML developer reference.** XXX: Microsoft Press, 2011.

DAVIS, A. M. **Software requirements:** objects, functions and states. : Prentice Hall, 1993.

GOGUEN, J. A. Techniques for Requirements Elicitation. In: **Software Requirements Engineering.** XXXX: IEEE Computer Society Press, 1997.

HARJONO, J. H. **XUL Application Development.** Singapore: Nanyang Technological University, 2001.

JACKSON, M. **Software Requirements and Specifications.** London: Addison-Wesley, 1995.

KOTONYA, G., SOMMERVILLE, I. **Requirements Engineering Processes e Techniques.** XXXX: John Wiley and Sons, 1998.

Lamsweerde, A., Darimont, R. Letier, E. Managing Conflicts in Goal - Driven Requirements Engineering. In: **IEEE Transactions on Software Engineering.** Special Issue on Managing Inconsistency in Software Development. XXXX: YYYY, 1998.

LAMSWEERDE, A. V. **Requirements engineering in the year 00:** a research perspective. Limerick: ACM Press, 2000.

LEE, W. J., CHA, S. D., KWON, Y. R. Integration and analysis of use cases using modular petri nets in requirements engineering. In: **IEEE Transactions on Software Engineering.** Washington: IEEE Computer Society, 1998.

LEITE, J. C. S. P. **Engenharia de Requisitos:** Notas de Aula. Rio de Janeiro, PUC-Rio, 1994.

PFLEEGER, S. L. **Engenharia de software:** teoria e prática. São Paulo: Prentice Hall, 2004.

LOUCOPOULOS, P., KAVAKLI, V. **Enterprise Modelling Engineering, International and Teleological Approach to Requirements Engineering Journal of Intelligent and Cooperative Information Systems.** XXXX: YYYY, 1995.

KOTONYA, G., SOMMERVILLE, I. **Requirements Engineering:** Processes and Techniques. West Sussex: John Willey & Sons Ltd, 1998.

SOMMERVILLE, I. **Software engineering.** XXXX: Addison-Wesley, 2004.

SOMMERVILLE, I., SAWYER, P. **Requirements engineering:** a good practice guide. XXXX: Wiley, 1997.

THAYER, R. H., DORFMAN, M. Introduction to Tutorial Software Requirements Enginnering. In: **Software Requirements Engineering.** Washington: IEEE Computer Society, 1997.

ZAVE, P. Classification of research efforts in requirements engineering. In: **ACM Computing Surveys.** New York: ACM Press, 1997.

# 3

# Identificação e especificação de componentes

# Identificação e especificação de componentes

## Introdução

Reutilizar componentes de *software* pode ser considerado um fator-chave para aumentar a qualidade e produtividade no desenvolvimento de sistemas. Já que esses componentes foram codificados e testados, dessa forma há a redução do tempo de desenvolvimento, de testes e das possíveis correções de falhas no produto de *software*. Conforme já tratado no Capítulo 1, componentes podem ser definidos como módulos que se utilizam de interfaces para que ocorra a conexão entre eles e entre eles e o sistema, colocando assim o foco de possíveis adaptações nos componentes em suas interfaces.

Sendo assim, a utilização de componentes permite que códigos já escritos sejam reutilizados, reaproveitando dessa forma a arquitetura de um componente de determinado sistema em outro sistema de *software*. E, para que haja essa alocação a outro sistema, o componente que será reutilizado passará por alterações a fim de atender de forma específica aos requisitos do *software* ao qual será integrado.

Portanto, é importante identificar e especificar quais são os componentes que serão necessários no desenvolvimento de *software*, saber onde encontrá-los, compreender como funciona o processo de identificação, quais são as características dos componentes que podem ser impostas na hora de especificá-los. Torna-se importante, também, conhecer tecnologias de componentes, como é o caso da tecnologia EJB, as metodologias utilizadas no desenvolvimento de código-fonte reutilizável e como é realizado o empacotamento, além de algumas técnicas de implementação de componentes.

Diante disso, neste capítulo estudaremos pontos relevantes para o entendimento sobre a identificação e especificação de componentes, além dos outros pontos citados acima.

## OBJETIVOS

- Compreender o que é a identificação de componentes;
- Conhecer o processo de identificação de componentes;
- Compreender a especificação de componentes;
- Entender os tipos de componentes EJB;

- Conhecer metodologias que utilizam componentes de software;
- Compreender a importância de empacotar componentes;
- Conhecer algumas técnicas de implementação de alterações em componentes.

## Identificação de componentes

Para Cattoni (1996), identificar componentes é uma tarefa que **se trata de analisar um sistema já desenvolvido com o objetivo de extrair possíveis componentes reutilizáveis**, em que se torna possível reaver o conhecimento, a experiência e o trabalho que foi aplicado em sistemas já desenvolvidos. No que diz respeito a identificar um código reutilizável, temos duas diferentes técnicas: as baseadas em transformação de componentes que já existem em componentes reutilizáveis e as baseadas na identificação de componentes por meio de métricas.

O primeiro procura eliminar dependências que são externas a um componente: aqui temos a análise de domínio e a reengenharia. A análise de domínio, de acordo com Araújo (1995), trata-se de um processo para identificar, coletar, organizar, analisar e representar determinado modelo de domínio e a arquitetura do *software* através de estudo de sistemas já existentes. Tal análise possui um papel importante na classificação do *software*, pois dessa forma é permitido que a arquitetura e os diversos relacionamentos entre componentes possam ser descritos. Já a reengenharia, segundo Furlan (1995), trata-se de uma série de técnicas que são orientadas a avaliar, reposicionar e transformar sistemas já existentes para aumentar a vida útil e permitir melhorias em sua qualidade funcional e técnica.

O segundo busca coletar todos os possíveis candidatos a componentes para a aplicação de uma série de métricas que têm por objetivo verificar seu potencial de reutilização. Tais métricas dão importância à análise de alto volume de código ao mesmo passo em que fornecem uma fase da análise automatizada. Com o intuito de simplificar o processo de extração em diversos ambientes, as métricas possibilitam o *feedback* e as melhorias.

## Processo de identificação de componentes

De acordo com Arnold (1994), esse processo se divide basicamente em duas etapas. Primeiramente, se **escolhe quais serão os componentes candidatos para a reutilização**; logo após, algum analista, com conhecimento voltado ao domínio da aplicação, poderá **analisar e determinar a utilidade de determinado componente.** Posteriormente, os componentes e suas respectivas informações são guardadas (ver Figura 1).

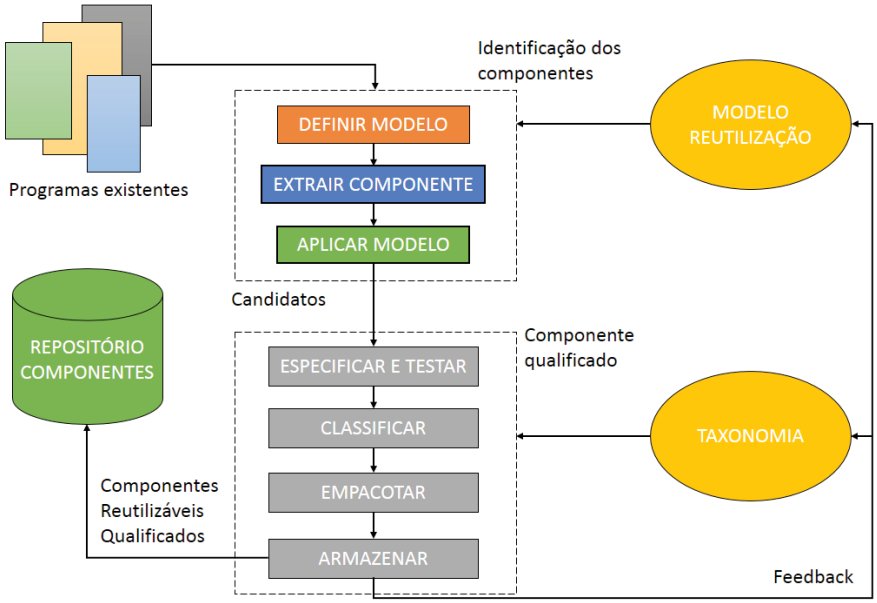


Figura 1. Etapas do processo de identificação de componentes. Adaptado de Arnold (1994).

Em toda a primeira etapa do processo de identificação de componentes, a quantidade de análises que devem ser realizadas por humanos é reduzida, tornando possível a automatização em sua totalidade, pois existe a limitação acerca dos componentes válidos para as análises. Mas, na segunda etapa, faz-se necessária a utilização de mão de obra humana em vez de realizar a procura por componentes reutilizáveis de forma direta em aplicações.

Nesse processo, a extração automática de componentes é realizada de forma independente e medida conforme as propriedades que foram observadas e

relacionadas ao seu grau de reuso. Com isso, podemos destacar três passos da identificação de componentes:

- **Definição de modelos de reusabilidade:** uma série de medidas automatizadas, que utilizam propriedades de determinado componente com o potencial de ser reutilizável, deve ser definida. Essas medidas têm de capturar métricas e valores utilizando as orientações que estão nos passos seguintes e modificá-las com o objetivo de alcançar modelos de reuso que estejam aptos para ampliar as chances de escolha de componentes candidatos para serem reutilizados;
- **Extração de componentes:** extrair componentes e complementá-los com o objetivo de que sejam utilizados de forma independente;
- **Aplicação do modelo:** o modelo definido é aplicado aos componentes que foram extraídos. Os candidatos a componentes para futuras análises são aqueles que estão dentro de determinados limites de reusabilidade definidos no modelo.

## Definição de especificação de componentes

Como já foi tratado no Capítulo 1 deste livro, a especificação de componentes é a especificação de uma unidade do *software*, na qual é descrito o comportamento de uma série de objetos do componente e definida a unidade responsável pela implementação. Tal comportamento é definido como um conjunto de interfaces.

Ou seja, a especificação se trata de um **levantamento de características necessárias aos componentes**. Suas características devem ser analisadas e separadas em grupos quando apresentarem propriedades comuns. Alguns exemplos dessas características podem ser: identificação, uso, maturidade, documentação, tecnologia, alterações e controle de qualidade. Para melhor entendimento, descrevemos essas características tratando-se de um mesmo componente:

1. **Identificação:** identificam os componentes.
   **Nome:** nome de identificação para o componente.
   **Origem:** recebe a identificação e contato do responsável por desenvolvê-lo (desenvolvedor, equipe, organização etc.);
2. **Uso:** aplicação ou contexto de utilização do componente.
   **Propósito:** funcionalidade do componente e para qual(is) problema(s) ele é a solução.
   **Domínio de aplicação:** contextos em que o componente pode ser empregado, bem como os sistemas em que ele pode ser integrado.

**Componentes similares:** componentes com a mesma finalidade.

**Tipo:** especificação (diagramas ou documentação) ou implementação (código-fonte ou executável).

**Fase de integração:** em qual fase do ciclo de vida o componente poderá ser integrado.

**Granularidade:** tamanho do componente (total de funcionalidade, número de casos de uso, fase de integração etc.);

3. **Maturidade:** grau de estabilidade e maturidade do componente.

   **Nível de reuso:** quantas vezes o componente foi utilizado em diversos sistemas.

   **Versões:** o controle de versionamento permite a reutilização e atualização ou correção de funções do componente;

4. **Documentação:** documentos do componente.

   **Modelo de componentes: e**stabelece os padrões (estrutura, recursos disponíveis etc.), descrevendo a maneira de interação entre as funcionalidades.

   **Especificação das interfaces:** as funcionalidades devem ser descritas, assim como as informações sobre as interfaces (nome, métodos, parâmetros, tipos etc.);

5. **Tecnologia:** informações a respeito da implementação e da tecnologia utilizada na construção do componente.

   **Infraestrutura:** suporte para a interação dos componentes de acordo com o modelo de componente, fornece a infraestrutura necessária para a execução dos componentes.

   **Portabilidade:** a plataforma para qual o componente foi desenvolvido.

   Ferramenta de desenvolvimento: a linguagem de programação, ambiente de desenvolvimento, compilador, biblioteca e outras ferramentas utilizadas.

   **Interoperabilidade:** a troca de informações entre diferentes componentes.

   **Interface gráfica:** ferramenta visual para manipular o componente.

   Características não funcionais: desempenho, segurança, confiabilidade etc.

   **Restrições:** as limitações devido a implementação ou tecnologia utilizada na construção do componente.

   **Distribuição:** identifica onde o componente está localizado e se ele pode ser distribuído (em rede);

6. **Alterações:** como as alterações podem ser realizadas no código-fonte do componente.

**Formas de modificações:** são identificadas as maneiras de modificar o componente.
**Acesso ao código-fonte:** identifica a permissão ao acesso do código-fonte do componente;

7. **Controle de qualidade:** controle e garantia de qualidade do componente.
**Métricas:** definição das métricas responsáveis pela avaliação da utilização, qualidade e maturidade.
**Teste:** mecanismos de testes de componentes.

**Tabela 1.** Exemplos de características levantadas para a especificação de componentes.

| CARACTERÍSTICAS | |
|---|---|
| IDENTIFICAÇÃO | Nome |
| | Origem |
| USO | Propósito |
| | Domínio da aplicação |
| | Componentes similares |
| | Tipo |
| | Fase de integração |
| | Granularidade |
| MATURIDADE | Nível de reuso |
| | Versões |
| DOCUMENTAÇÃO | Modelo de componentes |
| | Especificação das interfaces |

| | |
|---|---|
| TECNOLOGIA | Infraestrutura |
| | Portabilidade |
| | Ferramenta de desenvolvimento |
| | Interoperabilidade Infraestrutura |
| | Interface gráfica |
| | Características não funcionais |
| | Restrições |
| | Distribuição |
| ALTERAÇÕES | Formas de modificações |
| | Acesso ao código-fonte |
| CONTROLE DE QUALIDADE | Métricas |
| | Teste |

## Tipos de componentes

**O desenvolvimento de novos componentes é simplificado por meio da utilização de tecnologias de componentes** que existem em sua variedade. Neste item, iremos tratar sobre os tipos de componentes e a sua especificação mediante o uso da tecnologia EJB (*Enterprise JavaBeans*).

EJB é uma arquitetura de componentes voltada ao desenvolvimento e uso em sistemas corporativos baseados em componentes distribuídos (DEMICHIEL, 2000). Trabalha com a arquitetura cliente/servidor, na qual os componentes estão do lado servidor e podem ser acessados através do lado cliente por diversas ferramentas. **EJB também é definido como um modelo de componentes**, em que ele abrange as definições e especificações de um modelo que é flexível a fim de prover uma série de convenções, regras e definições voltadas à implementação de componentes através da linguagem de programação Java.

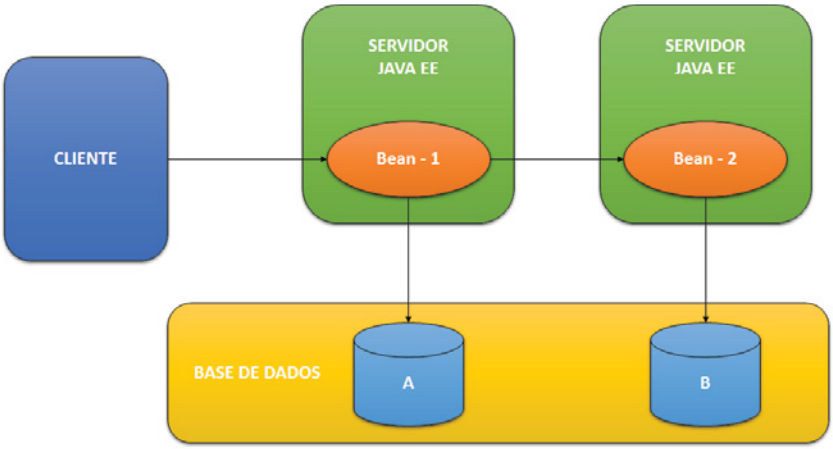


Figura 2. EJB e a arquitetura cliente/servidor.

As definições sobre os tipos de componentes EJB, sua infraestrutura, relacionamento de infraestrutura com componentes e entre os componentes, são realizadas por meio de contratos. Mas o EJB possui três tipos de componentes ou *beans: session bean, entity bean e message-driven bean*. Todos **eles estão presentes no lado servidor da arquitetura**, porém os dois primeiros são baseados em **RMI** (*Remote Method Invocation*) e são acessados por meio de protocolos de objetos distribuídos. Já o último apenas trata mensagens recebendo-as e as processando, baseando-se no **JMS** (*Java Message Service*).

## Session bean

O *Session bean* representa as extensões presentes no lado do cliente: a sua responsabilidade é a de executar tarefas; também chamadas de processos (MONSON-HAEFEL, 2001), elas podem ser acessos à base de dados. Sendo assim, esse tipo de componente **implementa as regras de negócios, a lógica, os algoritmos e os fluxos de trabalho** (Roman, 2002). O *Container* EJB, que é onde se disponibilizam os *beans* dentro da arquitetura EJB, instancia esses componentes quando há solicitação por parte do lado cliente. **O tempo de vida desses beans é curto** justamente por eles dependeram da execução quando solicitada pelo lado cliente.

Esses *beans* podem ser classificados de duas formas:

• ***Stateful:*** suas instâncias mantém o estado dos componentes entre as várias chamadas realizadas aos métodos pelo lado cliente. O fato de manter estado **permite a atualização de valores em suas variáveis enquanto o cliente estiver conectado**, e tais variáveis representam o estado de determinado cliente. Ao serem alteradas durante a execução de algum método, essas variáveis terão seus estados também alterados, e novas atualizações terão acesso a esses novos estados. **Exemplo de site de e-commerce:** de acordo com adições de um produto no carrinho de compras, o cliente realiza novas interações, e a cada uma dessas adições o estado é atualizado e mantido entre essas interações. Ou seja, o cliente pode clicar em "continuar comprando" que o carrinho de compras conterá todos os itens que tiveram cliques em "comprar";

• ***Stateless:*** diferentemente do *stateful*, o *stateless* representa somente uma requisição a cada interação com o lado cliente. Ou seja, **recebida a requisição, o componente finaliza o seu trabalho**. Nesse tipo de *beans*, não existe o ciclo requisições-respostas, portanto não se tem a necessidade de manter o estado do componente. Nesse caso, o *Container* EJB pode escolher entre destruir ou recriar o *bean*, após o método ser executado, e prepará-lo para futuras invocações. E, lógico, essa escolha depende do algoritmo codificado. **Exemplo de verificação de cartão de crédito:** o componente recebe uma série de informações (número de cartão, nome do titular, valor da compra etc.) e, dependendo da existência de crédito na conta, é retornada somente uma resposta. E, após essa resposta, o *bean* poderá receber outras solicitações, não armazenando as informações utilizadas nas solicitações anteriores. É por isso que muitas vezes temos de adicionar o número de CPF, RG ou até mesmo a leitura digital a cada transação (verificar saldo, transferência, depósito, por exemplo) que realizarmos.

### Entity bean

O *Entity bean* representa os componentes capazes de tornar dados permanentes em mecanismos de armazenamento (Roman, 2002). Eles **executam seus processos ou implementam fluxos de trabalho do sistema representando os dados e informações que estão armazenados em dispositivos**. Os objetos do negócio são representados por esses tipos de componentes, e em alguns casos esses componentes têm mapeamento voltado à tabela de base de dados relacional,

em que cada registro da tabela representa determinada instância. Outra diferença entre esse e o *bean* anterior está relacionada **ao ciclo de vida que, nesse caso, é mais longo**. O *Entity bean* **não possui dependência de sessão com o cliente, mas depende da existência de dados em sua base de dados** (Roman, 2002).

Devido ao *Entity bean* utilizar-se de acesso a dados presentes em uma base de dados de um dispositivo, devemos compreender que, para isso, deve haver **sincronização** feita de forma **periódica** entre o componente e o dispositivo. Tomando conhecimento disso, podemos classificar a persistência desses dados na tecnologia EJB, e essa classificação está relacionada à diferença entre os responsáveis por tal implementação. São elas:

- ***Bean-managed persistence:*** nessa classificação, o próprio *bean* é que tem a responsabilidade de garantir a persistência. O *container* EJB é quem define a realização da sincronização dos dados: ele lança os procedimentos para a realização da persistência;
- ***Container-managed persistence:*** já nessa classificação, quem realiza a persistência é o container EJB. O *container* EJB define tanto a execução quanto a sincronização propriamente dita. Os servidores EJB são responsáveis por providenciar os mecanismos a fim de dar a garantia automática da persistência.

### Message-driven bean

Esse tipo de componente se difere dos outros que, como já foi citado anteriormente, são baseados em RMI; portanto, eles são *beans* distribuídos que são acessados pela rede. Já o *Message-drive bean* **utiliza a tecnologia JMS, ou seja, recebe as mensagens que são enviadas por clientes JMS** (Roman, 2002). Sendo assim, nesse tipo de componente não temos os acessos através de invocação de métodos; aqui, esses acessos são realizados por meio de envio de mensagens. Uma característica desse *bean* é o fato de ele ser **assíncrono**, ou seja, os clientes JMS enviam as mensagens, mas não guardam as respostas dos *beans*, já que esses componentes não apresentam tipos de retornos.

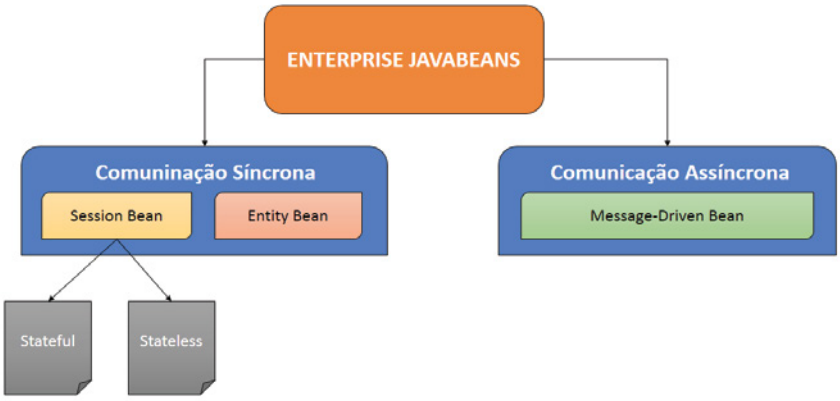


Figura 3. Enterprise JavaBeans.

## Metodologias/Padrões

A tecnologia de componentes de *software* é bastante utilizada para gerar suporte tecnológico, com seus conceitos e produtos, ao desenvolvimento e funcionamento de aplicações baseadas em componentes. De acordo com Bass et al. (2000), essa tecnologia auxilia na construção de aplicações seguindo uma mesma linha de padrão de arquitetura. Tecnologias como *Enterprise JavaBeans*, da Sun, e o COM+, da *Microsoft*, utilizão esse padrão. O *framework* de componentes é incluso na tecnologia de componentes, bem como outras ferramentas que são voltadas para *software* já desenvolvidos e baseados em componentes ou para projeto, construção, combinação e instalação desses componentes.

Conheceremos algumas metodologias/padrões que fazem uso da reusabilidade de *software*. São eles: Engenharia de Domínio, Desenvolvimento de *Software* Baseado em Componentes e, por último, mas não menos importante, Linha de Produtos de *Software*.

### Engenharia de Domínio

Processos voltados ao desenvolvimento de modelos, arquiteturas e componentes de determinado domínio compõem a Engenharia de Domínio. Basicamente, eles podem ser reusados em outros *softwares* que também fazem parte do mesmo domínio ou nicho. O processo de desenvolvimento de novos sistemas que fazem

parte do mesmo domínio é chamado de Engenharia de Aplicação. Ela utiliza-se de modelos, arquiteturas e componentes que são desenvolvidos na Engenharia de Domínio (ver Figura 4).

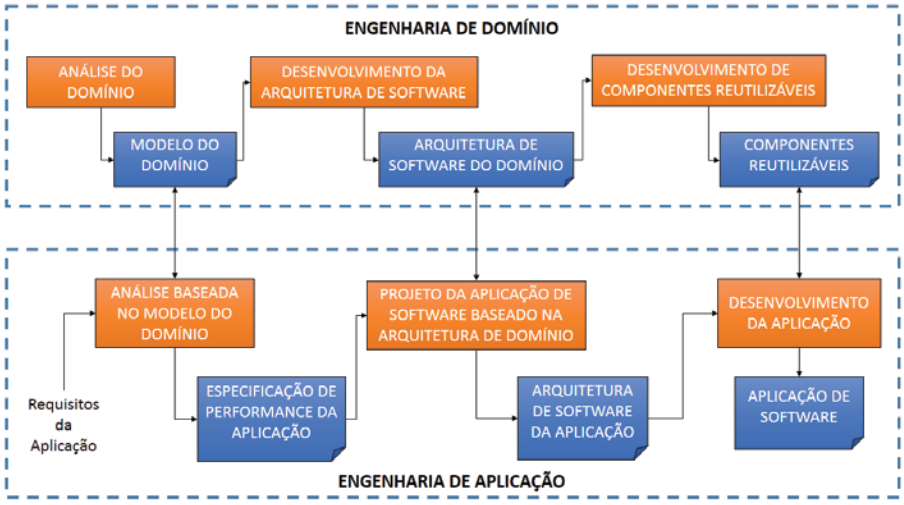


Figura 4. Ciclos de vida das Engenharias de Domínio e Aplicação.
Adaptado de (FOREMAN, 1996).

O suporte dado ao desenvolvimento de *software*, por parte da engenharia de domínio, **oferece respostas e recursos para que eles possam ser usados por uma linha de ferramentas pertencentes a um mesmo domínio**. Isso facilita o desenvolvimento de sistemas, a análise e predição comportamental. A atividade Análise de Domínio busca identificar e organizar informações acerca do domínio para que elas possam ser reutilizadas no decorrer do desenvolvimento de *softwares* que pertencem ao mesmo domínio. Como exemplos de metodologias pertencentes à engenharia de domínio, temos: *Draco, Feature-Oriented Domain Analysis e Organization Domain Model.*

Embora a engenharia de domínio não tenha apresentado resultados eficientes no final da década de 90, segundo Bayer et al. (1999), ela apresentou os fatores-chave responsáveis por isso: eles estavam relacionados às falhas ao definir o escopo de sistemas, a ausência de orientação operacional e a alta carga de foco nos problemas organizacionais.

Esse desenvolvimento é baseado em desenvolvimento de *software* iniciado na integração de componentes já existentes. Nesse contexto, os componentes de software, como visto no Capítulo 1 deste livro, podem ser desenvolvidos de forma interna pela organização ou adquiridos de terceiros, como os componentes de prateleira (COTS – *Commercial Off-the-shelf*). Eles, basicamente, podem ser comprados ou em alguns casos adquiridos de forma gratuita por possuírem licenças de código aberto. Esses últimos podem ser facilmente modificados por permitirem acesso ao código-fonte. Em ambos os casos, componentes de prateleira são disponibilizados na internet sob encomenda.

Observe que, c**om o desenvolvimento baseado em componentes, não existe a necessidade de codificação completa do sistema**, partindo do zero. Nesse desenvolvimento, existe apenas a integração de componentes no sistema. Dessa forma, **soluções de problemas comuns são reaproveitados**. É como se não precisasse se ater a soluções de problemas enfrentados por outros: eles desenvolvem suas soluções e as disponibilizam (COTS), permitindo que o foco ao desenvolvimento de outros sistemas esteja voltado à integração dos componentes (das soluções já existentes) e à codificação de determinadas funcionalidades da aplicação.

Na figura a seguir, são apresentadas as atividades que fazem parte do desenvolvimento de software baseado em componentes.

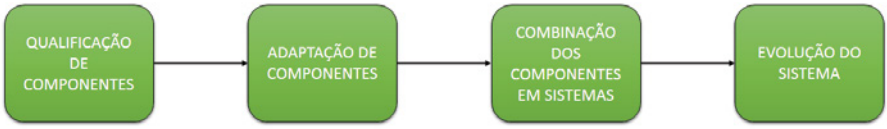


Figura 5. Atividades do Desenvolvimento de Software Baseado em Componentes.

• **Qualificação de componentes:** baseia-se em identificar quais são os componentes que mais se adequam e que possam auxiliar o desenvolvimento de *software*. Além disso, seleciona componentes entre as variadas alternativas que são disponibilizadas no mercado ou de forma gratuita;

• **Adaptação dos componentes:** mesmo existindo tecnologias que simplifiquem a integração de componentes no desenvolvimento de *software*,

às vezes existe a necessidade de realizar modificações nesses componentes para adaptá-los ao novo *software*;

- **Combinação dos componentes em sistemas:** a integração dos componentes no sistema utiliza a infraestrutura predefinida de acordo com a tecnologia de componentes de *software* que será utilizada;
- **Evolução do sistema:** corrigir erros de componentes pode resultar em substituí-los por outros ou até mesmo na codificação de novas soluções de tais problemas, que podem ser, por exemplo, modificações no comportamento dos componentes ou em suas interfaces.

## Linha de Produtos de Software

A Linha de Produtos de *Software* (SEI-CMU, 2016) trata-se de uma série de *softwares* e produtos deles que fazem parte de "um conjunto comum" com funcionalidades gerenciadas para satisfazer determinadas necessidades, presentes em um nicho de mercado, desenvolvidas "a partir de um núcleo de artefatos centrais comuns (core *assets*)" que seguem regras previamente determinadas. Ainda segundo o autor, compõem essa linha "artefatos reutilizáveis e recursos que formam a base para a linha de produto de software", além de abranger a arquitetura, componentes, modelos de domínio, requisitos, documentação, especificações, modelos de desempenho, cronogramas, planos de teste e seus casos, além das descrições de processo.

O *Software Product Line Practice* (Prática de Linha de Produtos de *Software*) é o "uso sistemático de artefatos centrais para montar, instanciar ou gerar os múltiplos produtos que constituem uma linha de produto de software" (SEI-CMU, 2016).

**Essa metodologia objetiva melhorar o tempo de produção, a qualidade e a manutenibilidade do *software* dentro de um nicho comum de mercado ou missão**. Embora possamos entender desenvolvimento de "linha de produtos" por um desenvolvimento feito a partir do zero em linha de produção, é justamente o contrário. Nessa metodologia, os softwares são desenvolvidos tomando como ponto de partida componentes já existentes dentro de um núcleo de artefatos centrais em comum, em que eles possam passar por modificações e ajustes de acordo dos quais o novo sistema necessita. As regras de arquitetura são definidas por esses artefatos centrais: tais regras devem, obrigatoriamente, ser seguidas por outros novos *softwares* da mesma linha de produtos.

Na figura a seguir, são apresentadas as atividades-chave, interativas e necessárias entre si, em linhas de produtos baseadas no desenvolvimento de artefatos centrais (*core asset development*), no desenvolvimento de produtos (*product development*) e no gerenciamento de linha de produto (*management*).


Figura 6. Atividades-chave em uma linha de produtos. Adaptado de (SEI-CMU, 2016).

- **Desenvolvimento de Artefatos Centrais:** é a atividade que alimenta o desenvolvimento de produtos baseados em componentes, a definição de regras e os recursos que podem ser reutilizados. Essa atividade recebe as necessidades e evoluções de artefatos vindo das atividades de desenvolvimento de produtos;
- **Desenvolvimento de Produtos:** desenvolve novos produtos de acordo com artefatos centrais comuns que estão disponíveis ou, ainda, a atualização desses artefatos para atender ao desenvolvimento de novos produtos;
- **Gerenciamento de Linha de Produtos:** possui um papel importante na tomada de decisão no que diz respeito ao investimento em recursos para desenvolver e manter os artefatos centrais.

## Empacotamento e implementação de componentes

### Empacotamento de componentes

Raramente, um componente será reutilizado em sua forma original; geralmente, será necessário alguma(s) alteração(ões) para adaptá-lo de acordo com

os requisitos do software que o irá receber. O empacotamento ou encapsulamento é uma das abordagens bastante utilizadas para **adaptar um componente ao novo sistema que o irá acoplar**, porém, em vez de modificá-lo, é criada uma nova visão externa para o componente.

Empacotar componentes é gerar uma nova visão para determinado componente, ou seja, **uma nova interface deverá ser adaptada**, diferentemente da interface original do componente, pois deverá atender a requisitos específicos para a nova reutilização (ver Figura 7).

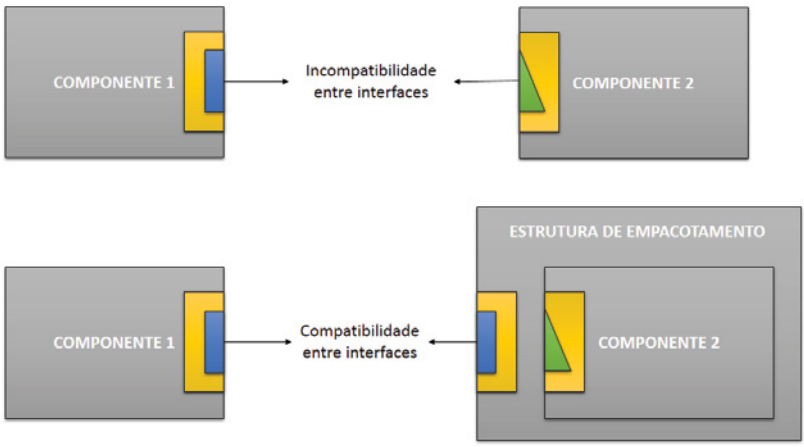


Figura 7. Empacotamento de componentes.

Na figura, podemos, assim, observar dois componentes (Componente 1 e Componente 2). As suas interfaces não são compatíveis entre si; dessa forma, não há como conectá-las no estado em que estão. Porém existe uma forma: observe que o segundo componente é acoplado a uma estrutura de empacotamento, e ele possui uma interface diferente da do segundo componente. E essa interface possui compatibilidade com a do primeiro componente; dessa forma, é possível conectar os componentes 1 e 2.

Essa abordagem de empacotamento de componentes pode ser utilizada para diversas finalidades. Uma delas é a **incompatibilidade de interfaces**, que geralmente se deve à assinatura de métodos diferentes, ou a linguagem de programação dos componentes é diferente, além da questão de plataformas etc. Dessa forma, a estrutura do empacotamento deve agir nesses pontos a fim de apresentar as suas soluções. Outra finalidade que podemos destacar aqui:

a **modificação de funcionalidade(s) original(is)** do componente que foi empacotado. Nesse acaso, a estrutura do empacotamento de componentes deve atuar na inserção de novas funcionalidades ou, ainda, modificar a forma original de tratar invocações de métodos do componente.

### Implementação de componentes

Ao implementar componentes já existentes, estamos tratando de modificá-los para determinada aplicação (Bosch, 1999). Conforme já tratado anteriormente, essas modificações ocorrem diante das necessidades de alterar determinado comportamento e funcionalidade de um componente. E, para isso, são realizadas inclusões, alterações ou remoções de funcionalidades. Ou, ainda, podem ser efetuadas modificações estruturais dos componentes para que seja permitida a sua reutilização em uma outra arquitetura diferente da sua original. E, por fim, essas modificações podem permitir que um único componente seja composto por outros componentes. Sendo assim, podemos destacar duas motivações para que haja as modificações em componentes: **incompatibilidade comportamental e a incompatibilidade estrutural**. A primeira está relacionada às diferenças das funcionalidades que se tem entre as que se necessita ter, enquanto a segunda se relaciona com o fato de quanto as interfaces fornecem e o quanto são requeridas. Essas duas motivações podem atuar unidas.

Nesse momento, três termos podem ser utilizados na implementação de componentes:

- **Adaptação:** trata-se de modificações feitas por um desenvolvedor (programador) para que o componente possa ser utilizado de uma nova forma em outro sistema;
- **Customização:** trata-se de utilizar uma alternativa para o funcionamento do componente: por exemplo, reescrever uma funcionalidade em menos linhas ou, ainda, modificar a forma como um método recebe seus parâmetros;
- **Evolução:** trata-se de uma maneira de estender as propriedades e as funcionalidades de um componente.

Para tal, também existem algumas técnicas tradicionais que podem ser utilizadas. São elas (BOSCH, 1999): **técnicas de caixa-branca** (*white-box*) e **técnicas de**

**caixa-preta** (*black-box*). Essas técnicas serão descritas mais detalhadamente a seguir.

Tratando dos requisitos que devem ser observados e avaliados para a escolha de utilização de cada uma das técnicas, temos:

- **Transparência:** esse requisito pode ser dividido em outros três:
  - ✓ **Homogêneo:** o componente modificado deve ser utilizado conforme a sua forma original.
  - ✓ **Conservador:** deve ser mantida no componente modificado a sua parte original, fornecendo as suas atividades que não foram modificadas.
  - ✓ **Ignorante:** a existência do componente modificado deve não ser de conhecimento do componente original;
- **Componível:** não deve ser necessário que o componente seja redefinido, ele deve ser composto por outros componentes de forma facilitada. Isso favorece a possibilidade de que o componente possa ser modificado utilizando diferentes formas e tipos de adaptações;
- **Reusável:** mesmo modificando o componente, ele deve ser reutilizável com as suas alterações;
- **Configurável:** permite que o componente seja útil e reutilizável graças às suas modificações, podendo ser aplicado em outros componentes;
- **Foco arquitetural:** deve possuir especificação arquitetural, especificação do componente original e do modificado;
- **Independente de *framework:*** o componente deve funcionar da mesma maneira independentemente do *framework* de componentes;
- **Independente de linguagem de programação:** o componente deve ser independente de qualquer linguagem de programação que for utilizada na codificação do componente; dessa forma, outras linguagens poderão ser utilizadas.

### Técnicas de caixa-branca

As técnicas de caixa-branca são responsáveis por definir que a adaptação de determinado componente possa ocorrer por meio de alterações, substituições ou, até mesmo, exclusões de **especificações internas do componente**.

As técnicas do tipo caixa-branca, em geral, possuem características específicas voltadas para o **paradigma de programação** ou, ainda, para as próprias **linguagens**

**de programação** que são usadas na codificação de componentes. Exemplos de técnicas de caixa-branca:

- **Alteração de código:** modificar diretamente o código-fonte do componente. Isso permite ao desenvolvedor do sistema efetuar todas as modificações que julgar necessárias ao componente. Para que essa técnica seja realizada, faz-se necessário o acesso ao código do componente e, ao mesmo tempo, se exige o conhecimento de forma detalhada sobre como o componente funciona de forma interna. A partir do momento em que esse conhecimento for adquirido, torna-se possível continuar com as modificações no componente;
- **Herança:** baseia-se na herança presente em orientação a objetos. Da mesma forma que lá, todas as propriedades internas ou parte delas devem ser disponibilizadas ao componente modificado. Em Java e C++, são utilizados modificadores de acesso (*private* e *protected*) que especificam o que será disponibilizado para as classes inferiores (também conhecidas como subclasses ou classes filhas), e em *Smalltalk* esses modificadores de acesso não existem. Um outro ponto importante sobre essa técnica refere-se ao fato de que o código continuará a existir no mesmo espaço. As alterações realizadas com essa técnica também são feitas diretamente no código-fonte do componente. Com isso, faz-se necessário possuir o conhecimento funcional das super e subclasses que deverão ser modificadas.

### Técnicas de caixa-preta

As técnicas de caixa-preta baseiam-se em reutilizar o componente em sua forma original, **modificando somente as suas interfaces**. Isso exige o completo entendimento acerca das interfaces dos componentes, já não dependendo das especificações internas para tal.

As técnicas do tipo caixa-preta, frequentemente, possuem características específicas relacionadas a determinados **padrões de projetos**. Exemplos de técnicas de caixa-preta:

- ***Wrapping:*** está relacionada com a definição do componente, cujo componente original é encapsulado e utilizado como filtro para que as requisições sejam recebidas, para, assim, definir qual será o comportamento do componente. A principal utilização dessa técnica é concentrada na solução de interfaces incompatíveis. Essa técnica não exige o conhecimento acerca do funcionamento interno do componente, pois as modificações são realizadas de forma externa

ao componente. Porém essa técnica está ligada à sobrecarga de codificação nas interfaces, incluindo aquelas em que não há necessidade de alterações;

- **Proxy:** essa técnica faz o intermédio entre a comunicação e o componente original. Sendo assim, as requisições recebidas são redirecionadas ao componente. Diferentemente da técnica anterior, essa não encapsula o componente original, mas fica alocada para filtrar as requisições que chegam para o componente original. Aqui é possível efetuar o acesso direto ao componente original: isso elimina possíveis sobrecargas de codificações. Não é exigido conhecimento interno sobre o funcionamento dos componentes, já que as modificações são feitas de forma externa por meio do componente *proxy*.

## ATIVIDADE

**01.** De acordo com o que foi estudado neste capítulo, conceitue identificação de componentes e cite quais são as técnicas utilizadas para identificar componentes.

**02.** De acordo com o que foi estudado neste capítulo, cite e conceitue os passos para identificar componentes.

**03.** De acordo com o que foi estudado neste capítulo, utilizamos a tecnologia EJB para exemplificar os tipos de componentes. Quais são esses tipos?

**04.** De acordo com o texto, qual a principal diferença entre o *Session bean* e o *Entity bean* com o *Message-driven bean*?

**05.** De acordo com o texto, cite os modelos para desenvolvimento que utilizam códigos reutilizáveis juntamente com suas respectivas atividades.

**06.** De acordo com o texto, o que é empacotamento de componentes e em qual contexto ele pode ser utilizado?

# GABARITO

**01.** Trata-se de uma tarefa que procura analisar um sistema já desenvolvido com o objetivo de extrair possíveis componentes reutilizáveis, em que se torna possível reaver o conhecimento, a experiência e o trabalho que foram aplicados em sistemas já desenvolvidos. Sobre as técnicas utilizadas para identificar código reutilizável, temos as baseadas em transformação de componentes que já existem em componentes reutilizáveis e as baseadas em identificar componentes por meio de métricas.

**02.** Os passos para a identificação de componentes são: definição de modelos de reusabilidade, extração de componentes e aplicação do modelo. A definição de modelos de reusabilidade trata-se de uma série de medidas automatizadas que utilizam propriedades de determinado componente com o potencial de ser reutilizável. Essas medidas têm de capturar métricas e valores utilizando as orientações que estão nos passos seguintes e modificá-los com o objetivo de alcançar modelos de reuso que estejam aptos para ampliar as chances de escolha de componentes candidatos para serem reutilizados. Na extração de componentes, busca-se extrair os componentes e complementá-los com o objetivo de que sejam utilizados de forma independente. Já na aplicação do modelo, o modelo definido é aplicado aos componentes que foram extraídos. Os candidatos a componentes para futuras análises são aqueles que estão dentro de determinados limites de reusabilidade definidos no modelo.

**03.** Ostipossão: *Session bean, Entity bean* e *Message-driven bean.*

**04.** Tanto o *Session bean* quanto o *Entity bean* são baseados em RMI (*Remote Method Invocation*); portanto, são acessados por meio de protocolos de objetos distribuídos. Já o *Message-driven bean* somente trata as mensagens, recebendo-as e as processando, com base no JMS (*Java Message Service*).

**05.** Os modelos estudados foram: Engenharia de Domínio, Desenvolvimento de *Software* Baseado em Componentes e Linha de Produtos de *Software*.

Suas atividades:

- Engenharia de domínio: Análise de domínio, Desenvolvimento de arquitetura de *software* e Desenvolvimento de componentes reutilizáveis;

▪ Desenvolvimento de *software* baseado em componentes: Qualificação de componentes, Adaptação dos componentes, Combinação dos componentes em sistemas e Evolução do sistema;
▪ Linha de produtos de *software*: desenvolvimento de artefatos centrais (*core asset development*), desenvolvimento de produtos (*product development*) e gerenciamento de linha de produto (*management*).

**06.** Empacotar componentes é gerar uma nova visão para determinado componente, ou seja, uma nova interface deverá ser adaptada, diferentemente da interface original do componente, pois deverá atender a requisitos específicos para a nova reutilização. Ela é uma das abordagens mais utilizadas para adaptar um componente ao novo sistema que o irá acoplar, porém, em vez de modificá-lo, é criada uma nova visão externa para o componente. O empacotamento de componentes é geralmente utilizado quando se torna necessário alguma(s) alteração(ões) para adaptar o componente de acordo com os requisitos do *software* que o irá receber.

## REFLEXÃO

Neste capítulo, iniciamos os nossos estudos acerca da Identificação e Especificação de Componentes. Começamos o nosso aprendizado com o conceito de Identificação de Componentes, em que fomos apresentados para duas técnicas e conhecemos o processo de identificação de componentes. Conceituamos a definição da especificação de componentes tomamos como exemplo características de um suposto componente e detalhamos tais características para compreendermos como ela funciona.

Conhecemos os tipos de componentes presentes na especificação de componentes de software a serem desenvolvidos por meio da tecnologia EJB. Conhecemos os seus tipos (*Session bean, Entity bean* e *Message-drive bean*) e suas diferenças. Também conhecemos três metodologias (Engenharia de Domínio, Desenvolvimento de *Software* Baseado em Componentes, Linha de Produtos de *Software*) e suas atividades que fazem uso da reusabilidade de *software*. Em Desenvolvimento de *Software* Baseado em Componentes, reforçamos o conceito de COTS (*Commercial Off-the-shelf*) visto no Capítulo 1 deste livro.

Conceituamos, também, a atividade de Empacotamento de Componentes, em que observamos que ela parte de uma nova interface que deverá ser adaptada, diferentemente da interface original do componente, pois deverá atender a requisitos específicos para a nova reutilização. Ou seja, será necessário alguma(s) alteração(ões) para adaptá-lo de acordo

com os requisitos do *software* que o irá receber. E podemos compreender isso através de exemplos de dois contextos para o empacotamento de componentes. Conhecemos também as técnicas de caixa-branca e caixa-preta, bem como suas respectivas técnicas que auxiliam na implementação de modificações em componentes.

Finalizamos este capítulo com uma base de conhecimentos solidificada acerca de identificação e especificação de componentes, o que complementa os estudos realizados nos capítulos anteriores. Com os conhecimentos adquiridos neste capítulo, podemos continuar os nossos estudos neste livro.

## LEITURA

Para você avançar mais no tocante a seu nível de aprendizagem envolvendo os conceitos referentes a este capítulo, consulte as sugestões abaixo:

SOUSA, F. M., ALENCAR, F. M. R., CASTRO, J. F. B. **O Impacto dos COTS no Processo de Engenharia de Requisitos**. Workshop em Engenharia de Requisitos, WER. Buenos Aires: XXXX, 1999.

PRADO, A. F., LUCRÉDIO, D. **Ferramenta MVCASE** – Estágio Atual: Especificação, Projeto e Construção de Componentes.. Rio de Janeiro: Sociedade Brasileira de Computação, 2001.

DIAS, K., BORBA, P. **Padrões de Projeto para Estruturação de Aplicações Distribuídas Enterprise JavaBeans**. Segunda Conferência Latino-Americana em Linguagens de Padrões para Programação, SugarloafPLoP. Rio de Janeiro: Universidade Federal do Rio de Janeiro, 2002.

SIMÃO, R. P. S., BELCHIOR, A. D. **Um Padrão de Qualidade para Componentes de Software**. I Simpósio Brasileiro de Qualidade de Software. Gramado: Sociedade Brasileira de Computação, 2002.

## REFERÊNCIAS BIBLIOGRÁFICAS


_____. **A Framework for Software Product Line Practice**. Disponível em: http://www.sei.cmu.edu/productlines/frame_report/rel_domains.htm. Acesso em: 08/08/2016.

ARAÚJO, S. P. C. **Uma abordagem de apoio à reutilização de componentes em ferramentas ORACLE**. Blumenau: Universidade Regional de Blumenau, 1995.

ARNOLD, R. S.; FRAKES, W. B. **Software Reuse and Reengineering**. Califórnia: Institute of Eletrical and Eletronicsengenieers, 1994.

BASS, L. et al. **Volume II:** Technical concepts of components-based software engineering. Pittsburg: Carnegie Mellon University, 2000.

BAYER, J. et al. **PuLSE:** A Methodology to Develop Software Product Lines. Symposium on Software Reusability (SSR). Los Angeles: ACM Press, 1999.

BOSCH, J. **Superimposition:** a computer adaptation technique. Information and Software Technology. XXX: YYYY, 1999.

CATTONI, S. M. **Protótipo para a identificação de código reusável em Dataflex.** Blumenau: Universidade Regional de Blumenau, 1996.

DEMICHIEL, L. G. **Enterprise JavaBeans Specification.** Version 2.0. Palo Alto: Sun Microsystems, Inc., 2000.

FOREMAN, J. **Product** Line Based Software Development – Significant Results, Future Challenges. In: Software Technology Conference. Salt Lake City: 1996.

FURLAN, J. D. **Reengenharia da Informação:** Do Mito à Realidade. São Paulo: MAKRON Books do Brasil, 1995.

MONSON-Haefel, R. **Enterprise JavaBeans**. Sebastopol: O'Reilly, 2001.

ROMAN, E., AMBLER, S., JEWELL, T. **Mastering Enterprise JavaBeans**. New York: John Wiley & Sons, 2002.

YACOUB, S., AMMAR, H., MILI, A. Characterizing a Software Component. In: **Proc. of Int. Workshop on Component-Based Software Engineering**, 2. Los Angeles: XXXX, 1999.

# 4

# Interação de componentes

# Interação de componentes

## Introdução

É de fundamental importância entendermos como ocorre a interação entre os componentes de um sistema para que possamos gerenciar de forma eficaz o desenvolvimento tanto da arquitetura de sistema quanto a sua implementação de fato. E, para que possa ser alcançada a qualidade do produto de software, alguns pontos devem ser levados em consideração: são os padrões de desenvolvimento, que, neste capítulo, serão apresentados, como o processo de desenvolvimento RUP e o padrão de arquitetura MVC. Estudaremos também como funcionam os sistemas de gerenciamento de versão, um tipo de sistema que é muito utilizado para o controle e gerenciamento de alterações em artefatos de projetos, e o próprio código-fonte da aplicação. São pontos interessantes que os futuros profissionais de TI devem dominar para estarem aptos a atender à demanda do mercado.

Diante disso, neste capítulo estudaremos as interações dos componentes e como chegar a atingir os seus objetivos, saber distinguir o que são e quais são as regras de negócio, além do funcionamento do ciclo básico e das características comuns dos sistemas de gerenciamento de versão, bem como as suas classificações. Também trataremos dos padrões RUP e do arquitetural MVC. Serão aspectos importantes que servirão como base para a melhor compreensão sobre a arquitetura de sistemas.

## OBJETIVOS

- Conhecer a interação de componentes;
- Conhecer os fatores essenciais para que os objetivos da interação de componentes possam ser alcançados;
- Aprender a definição de operações ou as regras de negócio;
- Aprender como funcionam os sistemas de gerenciamento de versão;
- Entender como são classificados os sistemas de gerenciamento de versão;
- Definir o processo de desenvolvimento RUP;
- Conhecer os elementos estruturais do RUP;
- Aprender o que é o padrão arquitetural MVC;
- Diferenciar o padrão de três camadas do MVC com base na comunicação.

## Interação de componentes

Devemos estar atentos que, ao desenvolver um sistema utilizando componentes, eles terão suas características técnicas e funcionalidades específicas e diferentes dos demais componentes. Nesse contexto de interação, podemos chamá-los de componentes híbridos, e tais componentes correspondem, geralmente, a três classificações em relação aos seus comportamentos: componentes síncronos, componentes assíncronos e componentes autônomos.

**A maioria dos componentes se encaixa no tipo de componentes síncronos**. É preciso fazer o uso de abstração com o intuito de sincronizar processos; exemplificando, o componente síncrono é um **semáforo**. Ao utilizá-lo, deve-se de forma obrigatória armazenar a resposta de serviço desse tipo de componente, já que, para continuar a execução de determinada tarefa, existe uma dependência com resultados da execução dos serviços. Na figura a seguir, podemos visualizar o comportamento de um componente síncrono.

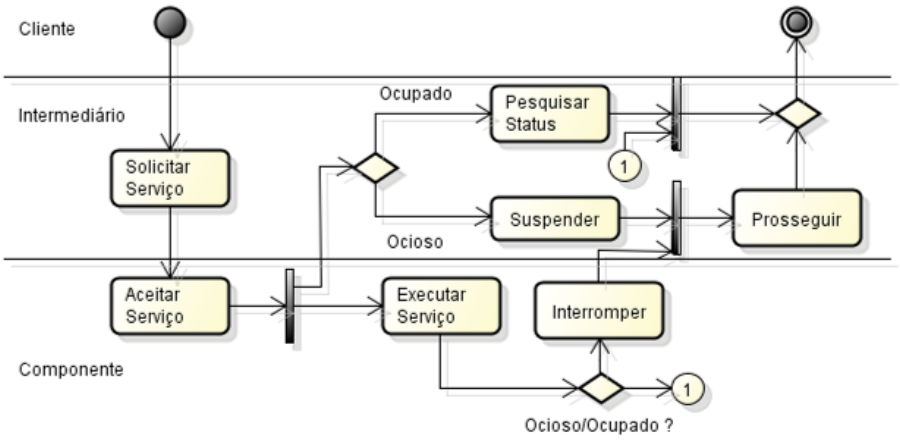


Figura 1. Comportamento de componente Síncrono.

A diferença do componente assíncrono em relação ao anterior é que ele não bloqueia determinado cliente à espera da finalização de seu serviço. Dessa forma, **o componente assíncrono permite que as tarefas sejam executadas de maneira concorrente**. Um exemplo é o recebimento de dados por uma rede. Geralmente, é realizada alguma notificação de sinaliza o término de determinado serviço com a utilização de funções de retorno ou, ainda, por meio da utilização de sistemas

que tratam de eventos. Na figura a seguir, podemos visualizar o comportamento de um componente assíncrono.

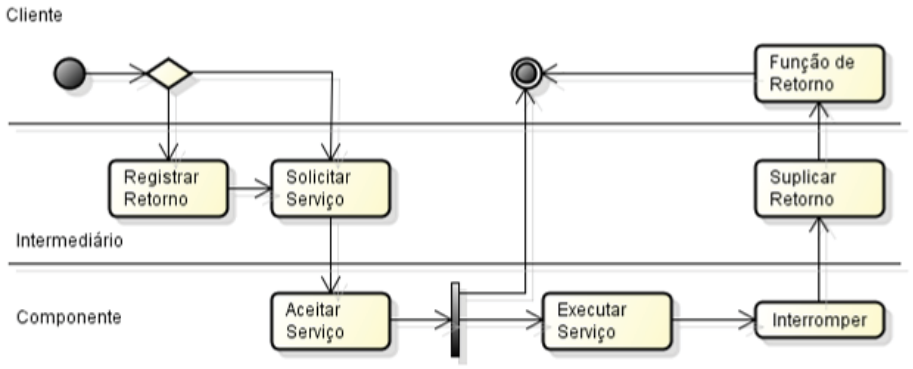


Figura 2. Comportamento de componente assíncrono.

Já **os componentes autônomos são aqueles que possuem serviços independentes de alguma chamada de componente**. Como exemplo, podemos citar um escalonador de tarefas ou, ainda, um gerenciador de energia. O que guia as tarefas do componente autônomo são eventos; no caso do escalonador de tarefas, podemos citar o fato de ele ser guiado por temporizador, enquanto o gerenciador de energia o é por outros componentes que estão relacionados com o monitoramento do estado da fonte de energia. Na figura a seguir, podemos visualizar o comportamento de um componente autônomo.

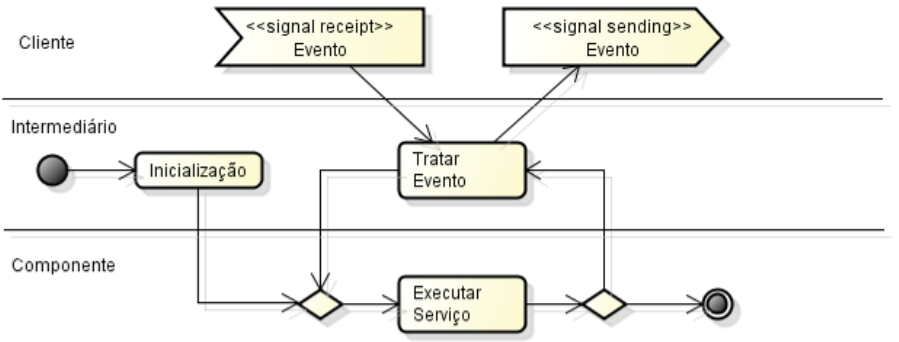


Figura 3. Comportamento de componente autônomo.

É importante a existência de mecanismos para a utilização de componentes que, em sua grande parte, possuem implementação, características e funcionalidades diversas. E, como já tratamos em capítulos anteriores. **A interação entre componentes é dada por chamadas a métodos de interface**. Essas interfaces, segundo (GIMENEZ, HUZITA, 2005), são responsáveis por agrupar uma série de serviços que estão relacionados, garantindo, assim, que tanto os dados quanto os processos estejam encapsulados.

Por meio das interfaces que conectam os componentes entre si, é possível habilitar a interação ou comunicação entre cliente e componente sem o acesso direto ao componente, apenas por interface. De forma geral, podemos imaginar os componentes possuindo dois tipos de interfaces para que haja a interação com os demais. Na figura a seguir, são apresentados esses dois "tipos" de interfaces que permitem a interação de componentes.

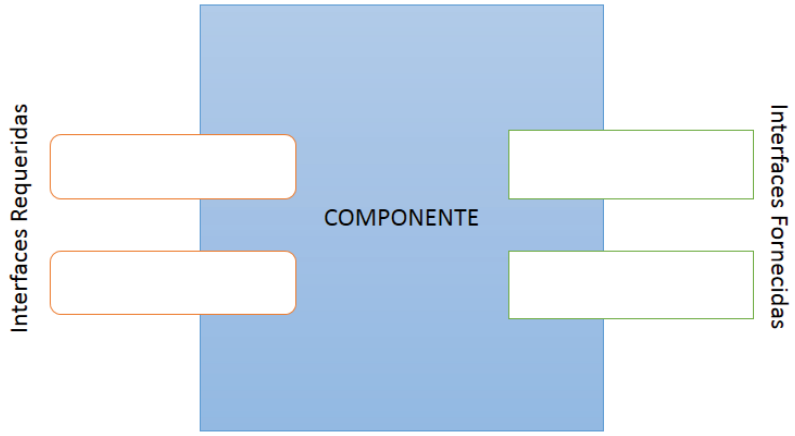


Figura 4. Interfaces de componentes. Adaptado de (SOMMERVILLE, 2003).

As interfaces fornecidas são responsáveis por definir quais são os serviços ou funcionalidades que o componente fornece. O componente fornece algum serviço para o sistema onde se encontra e/ou ainda algum serviço para que seja utilizado por outro componente. E as interfaces requeridas tratam de definir quais são os serviços que o componente precisa que sejam fornecidos a eles: são os serviços que o sistema precisa disponibilizar para que o componente possa funcionar corretamente. Sendo assim, podemos observar que o sistema também deve fornecer serviços para que os componentes possam funcionar adequadamente. Para Gimenes e Huzita (2005), não existe a necessidade de interface do tipo requerida quando o componente é autocontido.

A interação de componentes ocorre no sistema logo após a fase de especificação de componentes. Segundo Pagano (2004), operações com assinaturas completas são descobertas por meio de modelos de interação, e, somado a isso, as interfaces dos componentes são refinadas, divididas ou, ainda, agrupadas. Barroca, Gimenes e Huzita (2005) ressaltam que nessa fase de interação de componentes ocorre a identificação de novas operações relacionadas a negócios, já que os diagramas de colaboração UML são utilizados para detalhar suas interações.

A análise sobre interação de componentes tem como ponto de partida (BARROCA, GIMENES, HUZITA, 2005):

- Conjunto de interfaces de negócio;
- Conjunto de interfaces de sistema;
- Especificação inicial dos componentes e suas respectivas arquiteturas.

Nascimento (2005) pontua quatro fatores essenciais para que os objetivos da interação de componentes possam ser atingidos, e eles devem serem seguidos na seguinte ordem:

- **Desenvolver modelos de interação para cada operação do sistema:** análise dos casos de uso e verificação de quais passos irão precisar de funcionalidade no sistema. Para que essas funcionalidades ou operações sejam descobertas, se faz necessária a utilização de diagramas de colaboração do UML;
- **Descobrir operações das interfaces do sistema e suas assinaturas:** ainda com a utilização do diagrama de colaboração UML, existe a possibilidade de efetuar a identificação de funcionalidades que se fazem necessárias, possibilitando a especificação de assinaturas dessas funcionalidades;
- **Refinar responsabilidades:** as responsabilidades relacionadas aos tipos de negócios de forma não obrigatória podem ser refinadas logo após as descobertas das funcionalidades das interfaces e de suas respectivas assinaturas;
- **Definir restrições necessárias:** nesse passo, são definidas as pré e pós-condições para as funcionalidades da aplicação. Possui papel fundamental na definição de regras de negócio.

## Definir operações de negócios

Segundo o BRG (2001), para definirmos as operações ou regras de negócios, deve-se levar em consideração dois aspectos: **sistemas de informação e negócio**.

No primeiro, as regras de negócios são responsáveis pela definição de restrições devido a algum fator do negócio que se tem a preocupação de gerenciar. Nesse aspecto, as regras de negócios estão relacionadas com os dados que estão e poderão ser inseridos no sistema. Já no segundo, possuem a preocupação em tratar do comportamento do negócio propriamente dito. Assim, haverá todo um suporte para o gerenciamento e controle de riscos, ameaças ou oportunidades, além das políticas do negócio. Nesse aspecto, as regras de negócios envolvem pessoas ligadas ao negócio para que possam solucionar possíveis questões.

Ou seja, a abordagem utilizada pelo BRG (*Business Rule Group*) define as regras de negócios sob o ponto de vista de sistemas de *software* sem que haja algum tipo de compromisso ou dependência de tecnologias ou implementações. E sob a ótica do empreendimento sem qualquer presença de *software* ou tecnologias. Porém, embora exista essa distinção, na prática existe conflito: por muitas das vezes, esses dois aspectos possuem regras que atuam ao mesmo tempo.

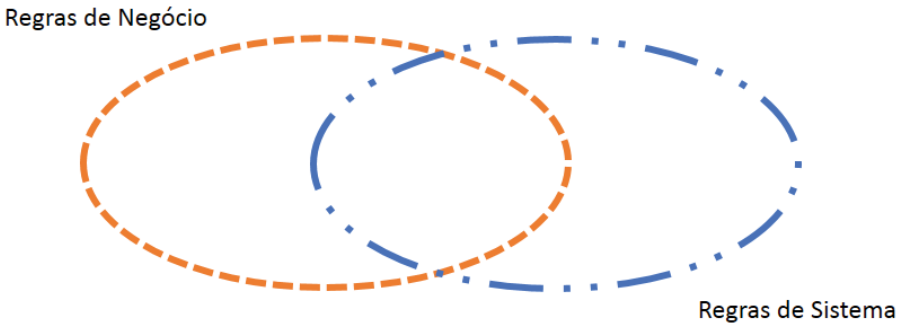


Figura 5. Regras de sistema e regras de negócio propriamente dito.

Um exemplo que podemos citar aqui é o seguinte: em determinada organização, existe uma regra que é responsável pelo cálculo do salário de seus funcionários que é de interesse tanto do departamento responsável pela aplicação que emite os pagamentos quanto do departamento responsável pela gerência de vendas. Nesse contexto, temos dois departamentos que precisam conhecer tal regra de negócio e aplicá-la. Porém também existem regras de negócio que são direcionadas a apenas um dos aspectos, como, por exemplo, regras voltadas para analisar a produtividade dos funcionários a fim de que sejam promovidos e, por outro lado, regras de negócio que permitem consultas otimizadas na aplicação da organização.

Existem, também, outras formas de entender e definir as regras de negócio. Para Ross (1998), **elas podem ser agrupadas de acordo com suas respostas a determinado evento, em que ele se trata de alteração de estado da aplicação ou alguma operação em execução**. Podem ser agrupadas em:

- **Regras de Rejeição:** determinado evento é rejeitado por não ser permitido. Exemplo: saldo em conta negativo; cadastro de e-mail já existe no sistema;
- **Regras de Projeção:** determinado evento gera outros eventos. Exemplo: caso saldo seja maior do que R$ 5.000,00, deve-se enviar panfletos sobre investimentos;
- **Regras de Produção:** não estão relacionadas a eventos, mas tratam de determinadas informações acerca do negócio. Exemplo: crédito menos débito é igual ao saldo da conta.

> *Regras do Negócio são declarações sobre a forma da empresa fazer negócio. Elas refletem políticas do negócio. Organizações têm políticas para satisfazer os objetivos do negócio, satisfazer clientes, fazer bom uso dos recursos e obedecer às leis ou convenções gerais do negócio. Regras do Negócio tornam-se requisitos, ou seja, podem ser implementados em um sistema de software como uma forma de requisitos de software desse sistema (LEITE, LEONARDI, 1998).*

## Sistemas de gerenciamento de versão

De acordo com Bolinger e Bronson (1995), o primeiro registro que se tem acerca do primeiro sistema de gerenciamento de versão data do ano de 1972: foi o *Source Code Control System* (SCCS), desenvolvido no laboratório *Bell Labs* por Marc. J. Rochkind. Até o RCS (*Revision Control System*) aparecer, o SCCS era considerado o principal sistema de gerenciamento de versão. Hoje, encontra-se obsoleto.

Walter F. Tichy desenvolveu o RCS, em 1982, por meio de métodos mais avançados do que apresentava o SCCS. Atualmente, o GNU Project ainda mantém o RCS. O RCS apenas gerenciava um arquivo de forma individual; diferente dele, o CVS (*Concurrent Versions System*) era capaz de gerenciar

projetos inteiros. O CVS teve seu início no ano de 1986 baseado no RCS; foi criado por Dick Grune, que inicialmente o tinha denominado de CMT. O CVS mais popular teve o seu desenvolvimento iniciado por Brian Berliner no ano de 1989. A empresa *CollabNet*, visando a melhorar o CVS, acabou desenvolvendo o *Subversion*, que também foi bastante aceito no mercado (SUSSMAN, FITZPATRICK, PILATO, 2009).

Os sistemas de gerenciamento de versão já citados são do tipo centralizado e de licença livre. Existem também os que são centralizados e de licença comercial, assim como há os do tipo distribuído, que também podem ser de licença livre ou comercial. Tanto que a SUN chegou a iniciar, na década de 90, o próprio sistema de gerenciamento de versões de forma distribuída, o *TeamWare*, que teria a responsabilidade de gerenciar os seus projetos internos. No entanto, o *TeamWare* foi deixado de lado, pois os desenvolvedores da SUN passaram a utilizar o *Mercurial*, que também é do tipo distribuído e com funções mais avançadas e modernas. O *Mercurial* foi desenvolvido em 2005 por Matt Mackall e teve uma enorme aceitação do mercado. Assim também ocorreu com o *Git*, que foi criado, no mesmo ano de 2005, por Linus Torvalds, e também teve uma grande aceitação do mercado. Ambos são sistemas distribuídos e de licença livre.

Na tabela a seguir, podemos observar diversos sistemas de gerenciamento de versão com seus respectivos anos de desenvolvimento, classificados em centralizado ou distribuído e em licença livre ou comercial.

Tabela 1. Sistemas de gerenciamento de versão.

| | | |
|---|---|---|
| CENTRALIZADO | LIVRE | SCCS (1972), RCS (1982), CVS (1990), CVSNT (1998), *Subversion* (2000). |
| | COMERCIAL | CCC/Harvest (1997), *ClearCase* (1992), *Sourcesafe* (1994), *Perforce* (1995), TFS (2005). |

| | | |
|---|---|---|
| COMERCIAL | LIVRE | GNU *arch* (2001), *Darcs* (2002), DCVS (2002), SVK (2003), *Monotone* (2003), *Codeville* (2005), *Git* (2005), *Mercurial* (2005), *Bazaar* (2005), *Fossil* (2007). |
| | COMERCIAL | *TeamWare*(199?), *Code co-op* (1997), *Bitkeeper* (1998), *Plastic* SCM (2006). |

Os sistemas de gerenciamento de versão possuem alguns pontos em comum:

- **Item de configuração:** elementos que são desenvolvidos no decorrer do desenvolvimento de *software* e que possuem identificação única, monitoramento e rastreamento de evolução;
- **Repositório:** local onde ficam armazenadas todas as versões;
- **Versão:** estado do item de configuração que está passando por suas possíveis alterações;
- **Revisão:** defeitos são corrigidos e/ou funcionalidades são implementadas;
- **Ramos:** são versões paralelas ou alternativas à principal;
- **Espaço de trabalho:** trata-se de um local onde é mantida uma cópia da versão que está sendo alterada, o que torna tal versão privada ao desenvolvedor responsável por alguma alteração;
- ***Check out:*** criação de cópia do trabalho presente no repositório;
- ***Update:*** atualizar o espaço de trabalho de acordo com as alterações armazenadas no repositório;
- ***Commit:*** criação de artefato ou atualização de sua versão que é salva do espaço de trabalho para o repositório;
- ***Merge:*** mesclagem entre diversas versões com o intuito de se criar uma única versão. Bastante utilizada quando vários desenvolvedores responsáveis por criar ou atualizar cada um em uma determinada funcionalidade devem unir os seus trabalhos para gerar uma única e mais completa versão;
- ***Changeset:*** coleção de modificações efetuadas nos arquivos que estão presentes no repositório.

**Dentro do repositório do sistema de gerenciamento de versão, é guardado o histórico de alterações que basicamente apresenta toda a evolução do projeto**. Os desenvolvedores devem efetuar uma cópia do repositório para que possam trabalhar em sua área de trabalho, aplicando o conceito de Update para poderem

realizar as alterações e salvá-las em sua área de trabalho para depois realizarem o *Commit*. Na figura a seguir, podemos visualizar melhor essa comunicação existente entre o repositório e a área de trabalho do desenvolvedor.


Figura 6. Comunicação entre repositório e área de trabalho. Adaptado de (DIAS, 2009).

Ainda sobre a classificação de sistemas de gerenciamento de versão entre centralizado e distribuídos, no centralizado temos a existência somente de um repositório central e diversas cópias do projeto. A figura 7 demonstra bem essa classificação.


Figura 7. Sistema de gerenciamento de versão centralizado. Adaptado de (DIAS, 2009).

Nesse modelo centralizado, o desenvolvedor deve efetuar um *check out* e, logo após, efetuar um *update* para o seu espaço de trabalho. Após as suas devidas

alterações, deve efetuar um *commit* e, assim, estará enviando as suas alterações para o repositório central. Porém, conflitos entre os arquivos são possíveis que ocorram entre os comandos de *commit* e de *update*. Uma solução para esse conflito é a utilização do comando *merge*.

Já o modelo distribuído, que também pode ser chamado de descentralizado, é composto por diversos repositórios independentes: cada desenvolvedor possui o seu somado a uma área de trabalho. Aqui, os comandos de *commit* e de *update* ocorrem de forma local. A figura 8 demonstra bem essa classificação.

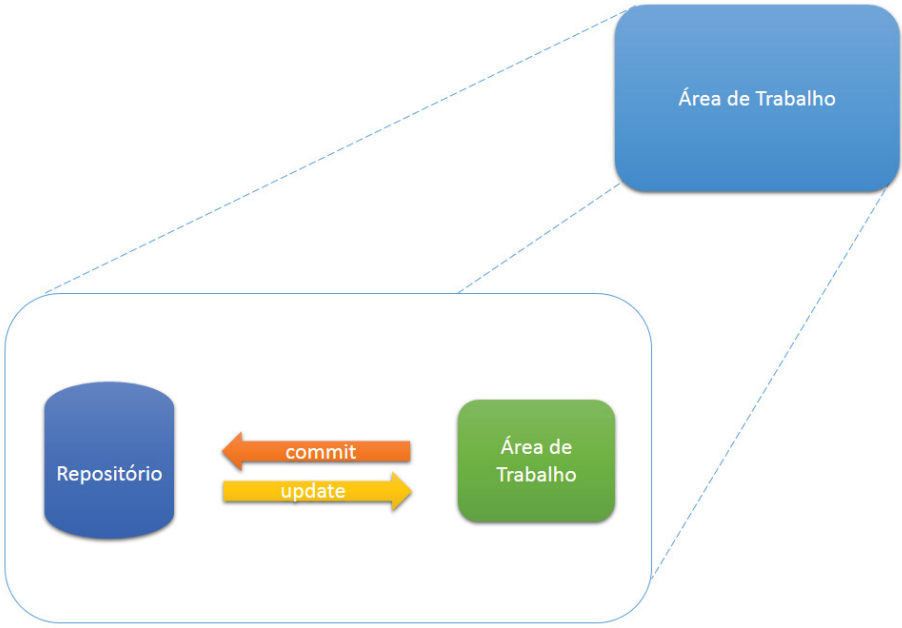


Figura 8. Sistema de gerenciamento de versão distribuído. Adaptado de (DIAS, 2009).

Nesse modelo, o desenvolvedor efetua os comandos da mesma forma que no modelo centralizado. As diferenças são: *check out* é denominado de *clone, commit* e *update* ocorrem localmente, a sincronização dos repositórios se dá pelos comandos *pull* e *push*. O primeiro atualiza o repositório local (destino) de acordo com as modificações efetuadas no repositório de origem. Já o comando *push* manda as modificações presentes no repositório local (origem) para o repositório de destino.

Existe a possibilidade de unir esses dois modelos de sistemas de gerenciamento de versão. Uma dessas formas é utilizar as ferramentas presentes no modelo distribuído

juntamente com a arquitetura cliente-servidor com um servidor (repositório) central. A figura a seguir demonstra bem a união dessas duas classificações.

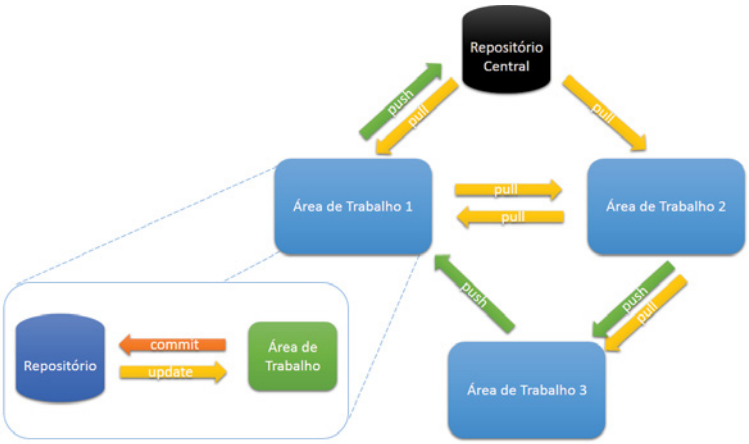


Figura 9. Exemplo de utilização em conjunto de sistemas de gerenciamento de versão distribuído e centralizado. Adaptado de (DIAS, 2009).

A seguir, iremos detalhar o ciclo básico de trabalho de três principais sistemas de gerenciamento de versão: *Subversion, Mercurial* e *Git*.

## Subversion

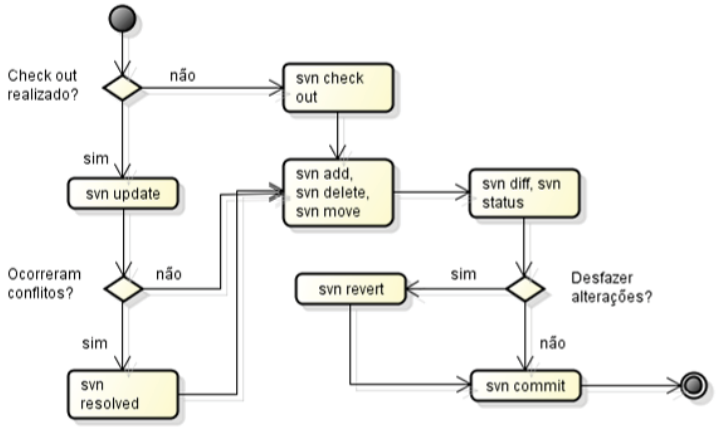


Figura 10. Ciclo básico de trabalho do Subversion.

• **Atualizar cópia de trabalho:** captura das modificações do repositório atualiza a cópia presente na área de trabalho por meio do comando *svn update*;

- **Realizar alterações:** *svn add* adiciona artefatos, *svn delete* exclui cópia, *svn copy* copia um artefato e *svn move* movimenta artefatos;
- **Verificar alterações:** as informações de estados de artefatos são exibidas pelo comando *svn status*, e o *svn diff* apresenta as diferenças presentes nas revisões;
- **Desfazer alterações:** *svn revert* reverte (desfaz) as modificações locais;
- **Resolver conflitos:** *svn update* atualiza no espaço de trabalho uma cópia, e o *svn resolved* remove o artefato que está em conflito;
- **Submeter alterações:** *svn commi*t envia ao repositório uma cópia com as alterações realizadas na área de trabalho.

O *Subversion* é portável entre os principais sistemas operacionais, como *Unix, Windows* e *Mac OS X*. Ele também possui interoperabilidade, possuindo funcionalidade nativa no *NetBeans* e disponibiliza *plug-ins* para diversas outras IDEs. E permite a utilização de protocolos de rede, como o HTTP e o HTTPS, ou algum outro protocolo próprio. Porém, implementá-lo não é o seu ponto forte se formos compará-lo com os outros sistemas de gerenciamento de versão que são mais modernos. O *Subversion* não disponibiliza o seu pacote completo para que seja instalado; dessa forma, existe um certo grau de trabalho para instalar todos os seus componentes.

### Mercurial

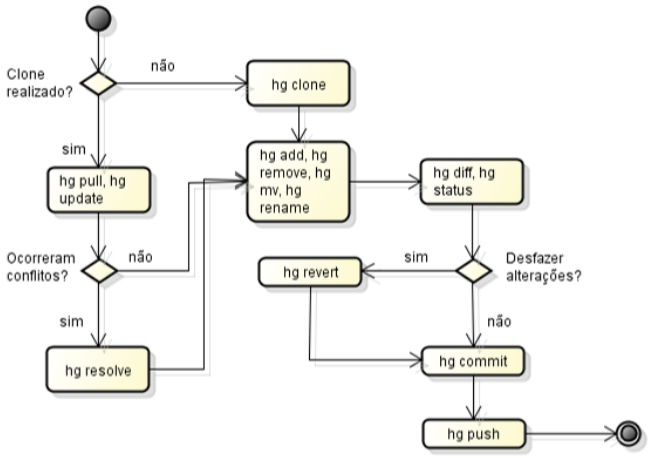


Figura 11. Ciclo básico de trabalho do Mercurial.

• **Atualizar repositório e cópia de trabalho:** *hg pull* efetua *download* das modificações do repositório, e o comando *hg update* mescla as modificações com a cópia de trabalho;

• **Fazer alterações:** *hg add* adiciona arquivos, *hg remove* exclui artefatos de uma cópia de trabalho, *hg mv* ou *hg rename* move ou renomeia artefatos;

• **Verificar alterações:** *hg status* mostra as informações de estados dos artefatos presentes na cópia de trabalho, *hg diff* apresenta as diferenças entre as revisões;

• **Desfazer algumas alterações:** *hg revert* reverte as alterações locais;

• **Resolver conflitos:** *hg resolve* os conflitos entre arquivos;

• **Submeter alterações:** *hg commit* manda as modificações realizadas na área de trabalho ao repositório;

• **Propagar alterações para outro repositório:** *hg push* manda as modificações locais para um repositório de destino.

O *Mercurial* é compatível com os sistemas *Windows, Linux, Mac OS X*, entre outros. A sua interoperabilidade permite disponibilizar plug-ins para diversas IDEs. Implementá-lo apresenta maior vantagem sobre o *Subversion*, já que disponibiliza pacote completo para que seja instalado. Também apresenta suporte a protocolos de rede, sendo eles HTTP e SSH.

### Git

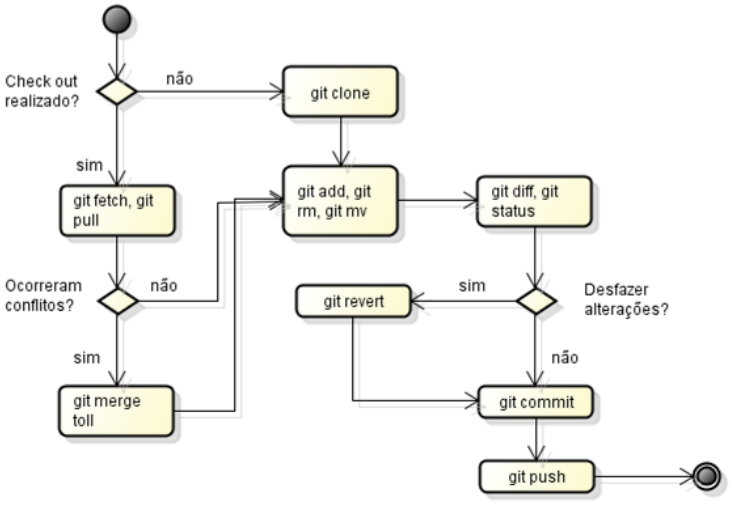


Figura 12. Ciclo básico de trabalho do Git.

• **Atualizar repositório e cópia de trabalho:** *git fetch* efetua *download* das modificações presentes no repositório, *git merge* mescla essas informações com o repositório local e *git pull* atualiza o repositório e o espaço de trabalho com as modificações presentes em outro repositório;

• **Fazer alterações:** *git add* adiciona artefatos, *git rm* exclui artefatos e *git mv* move ou renomeia artefatos;

• **Verificar alterações:** *git status* apresenta as informações de estado de artefatos, e *git diff* mostra as diferenças entre as revisões;

• **Desfazer alterações:** *git revert* reverte as modificações locais;

• **Resolver conflitos:** *git mergetool* resolve os conflitos de artefatos;

• **Submeter alterações:** *git commit* manda as modificações presentes no espaço de trabalho para o repositório;

• **Propagar alterações para outro repositório:** *git push* manda as modificações locais para um repositório de destino.

Embora tenha sido desenvolvido com foco em usuários *Linux, o Git* também possui suporte para outros sistemas operacionais, como *Unix, Windows, Mac OS X, BSD, Solaris* e *Darwin*. A interoperabilidade do *Git* permite a disponibilização de plug-ins para diversas IDEs. E, assim como o *Mercurial*, disponibiliza pacotes completos para instalação. Para os sistemas *Linux*, o *Git* é um pacote presente em seus repositórios de *softwares*. E também trabalha com protocolos de rede, como RSYNC, SSH, HTTP, HTTPS ou ainda protocolo próprio. O protocolo *Git* permite a otimização de largura de banda; dessa forma, as suas operações sobre as atualizações são realizadas de forma rápida e eficiente.

## Elementos do RUP

O RUP (*Rational Unified Process*) trata-se de um processo de desenvolvimento de *software* desenvolvido pela IBM *Rational Software Corportation*. É **um processo de alta qualidade repetível e previsível, estruturado com o intuito de desenvolver *software*** (KRUCHTEN, 2003).

A sua arquitetura está apresentada na figura a seguir.

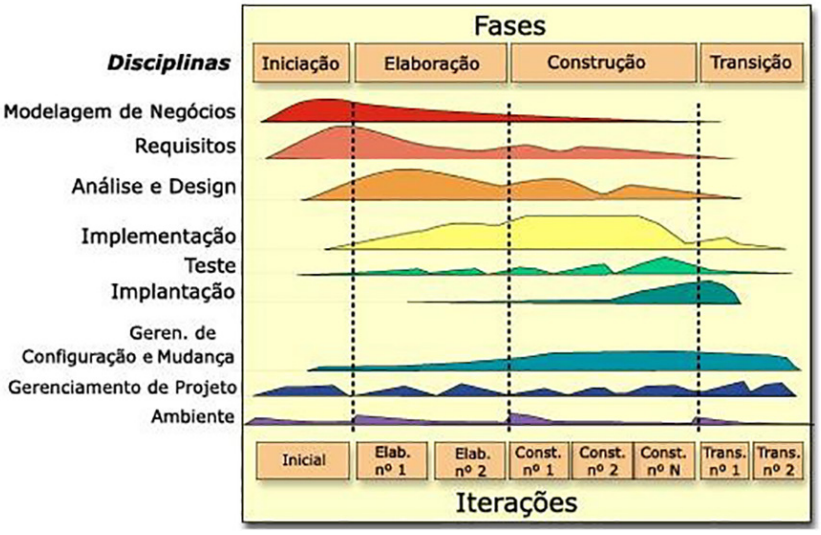


Figura 13. RUP – Gráfico de Baleias. Fonte: IBM – Metodologia RUP.

Nessa figura, podemos observar que a arquitetura do processo RUP é composta por duas dimensões: o eixo horizontal e o eixo vertical. O primeiro retrata o aspecto dinâmico, e nele é representado como acontece o desenvolvimento no decorrer do tempo com suas fases e interações, ou seja, apresenta o ciclo de vida do processo. Os esforços utilizados nas fases do RUP também estão apresentados (são eles que dão o nome de Baleias ao Gráfico), e podemos observar que nas interações iniciais é demandado mais esforço para a modelagem de negócio, os requisitos e a análise de projeto. E, nas interações finais, o esforço é mais demandado para a implementação, teste e distribuição. Já a segunda dimensão, o eixo vertical, trata-se de aspectos estáticos do RUP que, de forma tradicional, podem ser chamados de disciplinas.

Kroll e Kruchten (2003) apresentam três definições para o RUP:

1. Forma interativa de desenvolvimento de *software* guiado por casos de uso e centrado à arquitetura;

2. Processo bem definido e estruturado; nele, define-se o responsável pelas atividades, como elas devem ser realizadas e quando. O RUP apresenta estrutura voltada ao ciclo de vida de projeto;

3. Oferece estrutura personalizável para a engenharia de *software*.

O RUP pode se ser utilizado de diversas formas, pois não existe uma única maneira de aplicá-lo. Martins (2007) apresenta algumas características que diferenciam o RUP dos outros métodos interativos:

- Atacar riscos de forma antecipada e contínua;
- Certificação de que algo de valor é entregue ao cliente;
- Foco no *software* executável;
- Acomoda as mudanças antecipadas;
- Libera de forma antecipada um executável da arquitetura;
- Desenvolvimento de sistema com componentes;
- Trabalho em equipe;
- Qualidade como estilo de vida, não algo deixado para ser feito depois.

O RUP possui quatro elementos estruturais que correspondem a questões, como **"quem está fazendo o quê?"**, **"como?"** e **"quando?"**. São eles: **Papel (quem?)**, **Atividade (como?)**, **Artefato (o quê?)** e o **Fluxo (quando?)**. Vamos descrever cada um deles:

- **Papel:** aqui são definidas a questão comportamental e as responsabilidades atribuídas a um indivíduo ou equipe. O comportamento é descrito em atividades que serão executadas, e as responsabilidades são descritas em artefatos criados, modificados ou controlados pelo papel;
- **Atividade:** trata-se do trabalho que deve ser executado para que seja gerado um resultado significativo para o projeto. Ela pode estar relacionada tanto à criação quanto à atualização de artefatos. Para a sua execução, é designado um papel específico. Nesse elemento estrutural, pode ser utilizado o Mentor de Ferramenta, que, basicamente, fornece a forma de utilizar determinada ferramenta para executar a atividade;
- **Artefato:** trata-se de um produto, podendo ser um documento, modelo, código-fonte, programa executável, entre outros. Embora um artefato possa ser utilizado por vários papéis, apenas um papel é o responsável por criá-lo. Além disso, a sua produção, alteração e utilização são efetuadas nas atividades;
- **Fluxo:** está relacionado com atividades sequenciais que buscam a produção de resultado significativo. O fluxo é descrito através de diagrama UML de atividades. E é dividido em dois níveis: Fluxo Central e Detalhes do Fluxo.

## Padrão de arquitetura MVC

Inicialmente, o padrão arquitetural MVC (*Model View Controller*) foi desenvolvido na linguagem de programação *Smalltalk-76*, no ano de 1978, tendo como responsável o cientista, na época visitante no PARC (P*alo Alto Research Laboratory Xerox*), Reenskaug. O MVC foi criado com quatro componentes (*Model, View, Controller e Editor*) e com o intuito de resolver um problema no qual era dado aos usuários total controle sobre suas respectivas informações. O último componente era temporário e gerado pelo *View* sob demanda: ele aceitava entradas pelo mouse e teclado, era como uma interface que intermediava as ações entre os dispositivos de entrada e o *View*. Já por volta da década de 80, Jim Althoff e sua equipe criaram a nova versão do MVC implementada no *Smalltalk-80* com alterações no comportamento do componente *Model*. Com o passar do tempo, Reenskaug realizou algumas modificações, como a junção do *Editor* e *View*, além da aceitação e leitura para que o *Controller* e *View* pudessem ser montados juntos.

O padrão MVC se destaca por auxiliar os usuários na utilização de sistemas. Sendo assim, o usuário pode facilmente visualizar e manipular determinadas informações de forma intuitiva (REENSKAUG, 1978).

Uma das grandes vantagens do padrão arquitetural MVC para os desenvolvedores é a ajuda no desenvolvimento de sistemas de forma a haver uma **separação entre os componentes principais da aplicação**. Dessa forma, auxilia no armazenamento e na manipulação de dados, nas funções que possuem a responsabilidade de capturar os dados de entrada, além da visualização por parte dos usuários. Ou seja, o MVC determina o local para cada tipo de lógica dentro da aplicação (SANTOS et al., 2010). Esses tipos ou componentes são: ***Model, View e Controller***. O primeiro trata-se do objeto de aplicação; o segundo, da interface do usuário; e o último está relacionado com as entradas do componente *View* e como elas se comportarão (GAMMA et al., 2000).

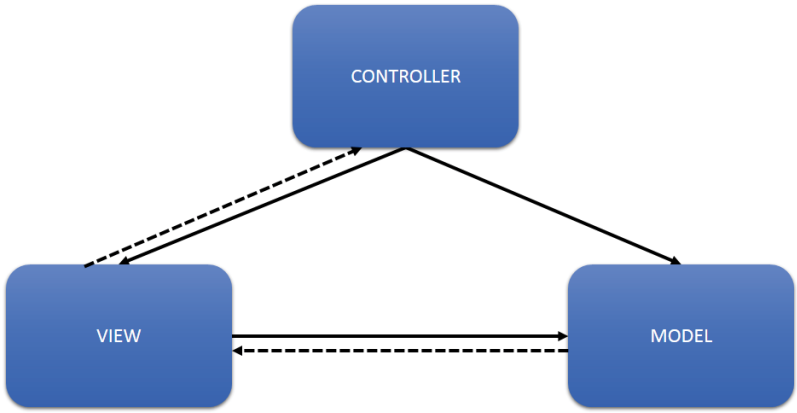


Figura 14. Arquitetura do padrão MVC.

O componente ***Model* é a camada lógica da aplicação**, representa os dados da aplicação e as suas regras de negócio; nele, estão contidas as operações necessárias para capturar e manipular dados presentes em um banco de dados, arquivo XML ou outros, para que esses dados possam ser visualizados no componente *View*, embora o seu comportamento em nada dependa do *View*. Tais operações incluem o CRUD (*Create, Reader, Update e Delete*). O *Model* fornece ao Controller a capacidade de acesso às operações/funções dele próprio. O componente ***View* é a camada de apresentação** responsável por gerar uma visualização dos dados e por recebê-los como dados de entrada dos usuários. O *View* não se importa de onde tais dados vêm, mas sim como eles serão mostrados. Ele também é responsável por enviar ao *Controller* os eventos de acordo com as ações do usuário e por acessar o Model via Controller para servir de interface com o usuário. Por último, mas não menos importante, o componente ***Controller* é a camada de controle da aplicação:** é dele a responsabilidade por gerenciar o fluxo da aplicação. Basicamente, ele trabalha com os dados inseridos no componente *View*, verificando qual das operações (exemplo, operações CRUD) será trabalhada no componente *Model*. A camada *Controller* é a que move a aplicação. Ou seja, o *Controller* interpreta as ações realizadas pelos usuários e as envia para o *Model*; assim, dessa forma, a aplicação segue determinado comportamento. Tais ações podem ser cliques em botões, interações com menus, opções de ativar/desativar funcionalidades etc. Dessa forma, o *Controller* pode, de acordo com as ações do usuário e do resultado do processamento no componente *Model*, selecionar qual das visualizações será usada e como ela será apresentada como resposta ao usuário no componente *View*. Sendo

assim, podemos observar que o *Controller* é basicamente uma camada intermediária que existe entre o *Model* e o *View*.

Na figura a seguir, está representado o padrão MVC com algumas operações e a sua comunicação entre cada um dos componentes (ZEMEL, 2009).

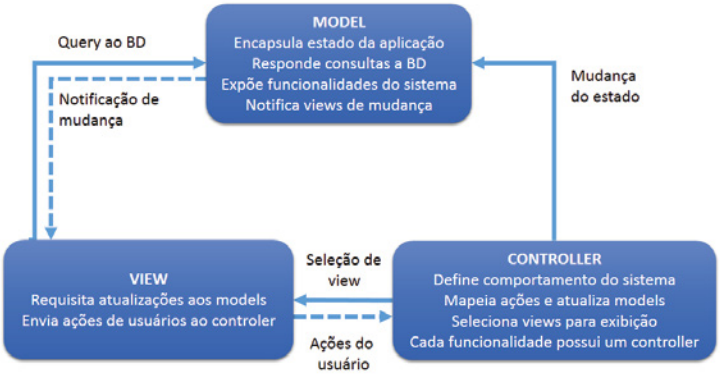


Figura 15. Comunicação do padrão MVC.

As vantagens de utilizar o padrão de arquitetura MVC estão relacionadas à sua capacidade de permitir gerenciar múltiplas visões utilizando somente um modelo, o que facilita prestar manutenção, testes e atualizar sistemas múltiplos. Facilita ainda a inserção de clientes, tem a capacidade de tornar o sistema escalável, possibilitando o desenvolvimento de forma paralela entre os componentes do MVC justamente por serem independentes. Porém, desenvolver com o MVC exige um grande tempo reservado somente para analisar e modelar a aplicação, e os desenvolvedores devem ser especialistas nesse contexto. E, também, a **utilização do MVC não é recomendada para sistemas com pequeno porte.**

Graças ao avanço dos computadores, da *Web* e dos sistemas dinâmicos, foram desenvolvidos diversos *Frameworks* que se utilizam dos conceitos de diversas arquiteturas; entre elas, o MVC (RIBEIRO, 2013). **Alguns exemplos de *Frameworks* MVC: *JSF, Spring MVC, Apache Struts, ASP.NET MVC, CakePHP, CodeIgniter, Laravel, Phalcon, Symfony, Zend Framework, Django, Web2py, Plone, Rails*, entre outros.**

Em resumo, **o padrão MVC possui um relacionamento com a arquitetura de sistemas e com a maneira como a mesma pode ser organizada pelos componentes ou camadas Model, View e Controller, além de como eles se comunicam entre si (MACORATTI, 2002). Vale ressaltar que existem diferenças entre a arquitetura** em três camadas e a arquitetura em MVC. Na arquitetura em três camadas, não é permitido à camada de apresentação se comunicar de forma direta com a camada de dados. Podem ser utilizadas camadas intermediárias para tal, porém essa comunicação somente é permitida se for realizada de forma linear e bidirecional. Ou seja, a informação é levada por somente um caminho, e nesse caminho são permitidas a ida e a vinda de informações. Já no padrão MVC é permitida a comunicação entre as camadas; e ela, por sua vez, acontece de maneira triangular. **O *View* sinaliza alguma modificação ao *Controller*, que, ao receber tal sinalização, atualiza a camada *Model*; assim, o *View* é atualizado de maneira direta a partir do *Model***. A Figura 16 apresenta bem essas diferenças. Deve estar claro que o **padrão de arquitetura MVC não se trata de uma arquitetura em três camadas** (MACORATTI, 2002).

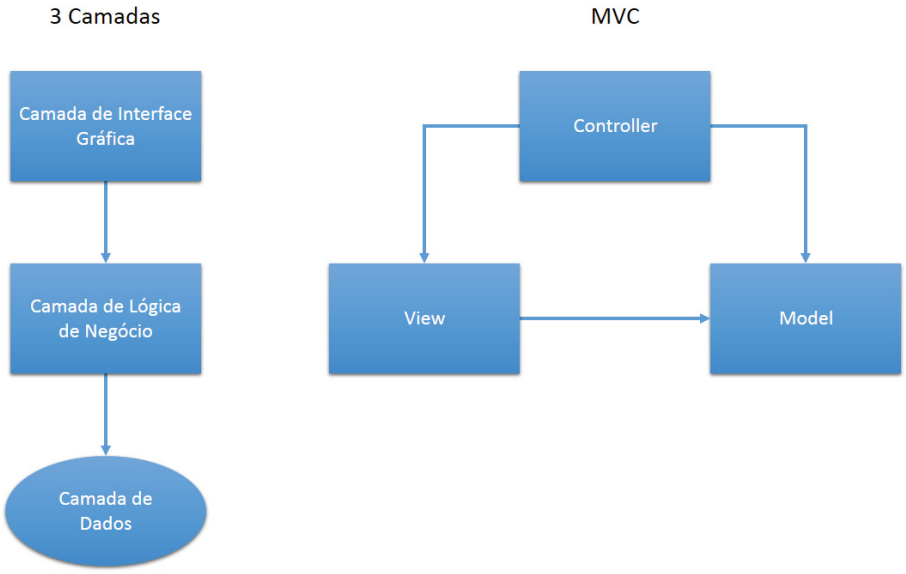


Figura 16. Diferenças entre a comunicação da arquitetura em camadas e a arquitetura em MVC.

## ATIVIDADE

**01.** De acordo com o que foi estudado neste capítulo, defina componentes síncrono, assíncrono e autônomo.

**02.** De acordo com o que foi estudado neste capítulo, diferencie sistemas de gerenciamento de versão centralizado e distribuído.

**03.** De acordo com o que foi estudado neste capítulo, defina o RUP e cite os seus elementos.

**04.** De acordo com o que foi estudado neste capítulo, o RUP apresenta algumas características que o diferenciam dos outros métodos interativos. Cite essas características.

**05.** De acordo com o que foi estudado neste capítulo, quais são os componentes do padrão arquitetural MVC e o que representa cada componente dele?

## GABARITO

**01.** Componente Síncrono: nele, é preciso fazer o uso de abstração com o intuito de sincronizar processos. Dessa forma, deve-se armazenar, de forma obrigatória, a resposta de serviço desse tipo de componente, já que, para continuar a execução de determinada tarefa, existe uma dependência com resultados da execução dos serviços.

Componente Assíncrono: ele não bloqueia determinado cliente à espera da finalização de seu serviço. O componente assíncrono permite que as tarefas sejam executadas de maneira concorrente. Nele, geralmente, é realizada alguma notificação que sinaliza o término de determinado serviço com a utilização de funções de retorno ou, ainda, por meio da utilização de sistemas que tratam de eventos.

Componente Autônomo: são aqueles que possuem serviços independentes de alguma chamada de componente. O que guia as tarefas do componente autônomo são eventos.

**02.** No sistema de gerenciamento de versão centralizado, temos a existência somente de um repositório central e diversas cópias do projeto. Já o sistema de gerenciamento de versão distribuído, que também pode ser chamado de descentralizado, é composto por

diversos repositórios independentes, e cada desenvolvedor possui o seu somado a uma área de trabalho.

**03.** O RUP pode ser definido como uma forma interativa de desenvolvimento de software guiado por casos de uso e centrado à arquitetura ou como um processo bem definido e estruturado, em que nele é definido o responsável pelas atividades, como elas devem ser realizadas e quando. O RUP apresenta estrutura voltada ao ciclo de vida de projeto ou, ainda, como um processo que oferece estrutura personalizável para a engenharia de software.

Os elementos do RUP são: papel, atividade, artefato e fluxo.

**04.** Algumas das características que diferenciam o RUP dos outros métodos interativos são:

- Atacar riscos de forma antecipada e contínua;
- Certificação de que algo de valor é entregue ao cliente;
- Foco no software executável;
- Acomoda as mudanças antecipadas;
- Libera de forma antecipada um executável da arquitetura;
- Desenvolvimento de sistema com componentes;
- Trabalho em equipe;
- Qualidade como estilo de vida, não algo deixado para ser feito depois.

**05.** Os componentes do padrão arquitetural MVC são: *Model, View* e *Controller*. O *Model* representa a camada lógica da aplicação; o *View*, a camada de apresentação; e o *Controller*, a camada de controle da aplicação.

## REFLEXÃO

Neste capítulo, demos início aos nossos estudos sobre a interação de componentes. Começamos por sua definição, em que tratamos de três tipos de componentes e dos quatros fatores essenciais para que os objetivos da interação de componentes possam ser atingidos. Em seguida, definimos as operações de negócio segundo alguns autores, quando pudemos dividir essas operações ou regras sob dois aspectos, classificá-las em grupo e ainda relacioná-las como requisitos.

Aprendemos também sobre sistemas de gerenciamento de versão, um pouco de sua história e a vasta implementação de aplicações com esse fim, além de alguns pontos em comum entre eles e a classificação que possuem. Como exemplo de sistemas de gerenciamento de versão, conhecemos como funciona o ciclo básico do *Subversion, Mercurial* e o *Git*, bem como alguns detalhes técnicos e os comandos desses três sistemas.

Continuamos os nossos estudos conhecendo os elementos do processo de desenvolvimento de *software* chamado RUP. Apresentamos três definições sobre esse processo, conhecemos o seu Gráfico de Baleias e ainda aprendemos algumas características que diferenciam o RUP dos outros métodos interativos. E também conhecemos e detalhamos cada um dos quatros elementos do RUP.

Ao estudarmos o padrão de arquitetura MVC, conhecemos um pouco de sua história e sobre a responsabilidade de cada um dos três componentes que constituem esse padrão de arquitetura. Conhecemos, também, suas vantagens e desvantagens, além de conhecermos alguns *frameworks* que fazem a utilização do MVC. E, ainda sobre MVC, aprendemos a diferença entre o MVC e a arquitetura em três camadas acerca da forma de comunicação que ambas possuem.

Finalizamos este capítulo com base sólida sobre a interação de componentes, as regras de negócio e, principalmente, sobre sistemas de gerenciamento de versão, processo de desenvolvimento RUP e o padrão arquitetural MVC. Complementam-se, assim, os estudos realizados nos capítulos anteriores deste livro. Sendo assim, poderemos dar continuidades ao próximo capítulo.

## LEITURA

Para você melhorar mais o seu nível de aprendizagem envolvendo os conceitos referentes a este capítulo, consulte as sugestões abaixo:


_____. **CakePHP Cookbook Documentation – Versão 2.x.** Disponível em: http://book.cakephp.org/2.0/_downloads/pt/CakePHPCookbook.pdf. Acesso em: 09/09/2016.

DUDLER, R. **Git** – guia prático. Disponível em: http://rogerdudler.github.io/git-guide/index.pt_BR.html. Acesso em: 31/08/2016.

CHACON, S., STRAUB, B. **Pro Git**. Disponível em: https://progit2.s3.amazonaws.com/en/2016-03-22-f3531/progit-en.1084.pdf. Acesso em: 31/08/2016.

ARAUJO, A. C. M., VASCONCELOS, A. M. L. Adaptando o RUP para o Desenvolvimento de Sistemas de Informações Web. In: **XI Conferência Internacional de Tecnologia de Software** – Qualidade de Software. Curitiba: XXXX, 2000.

BOENTE, A. N. P., OLIVEIRA, F. S. G., ALVES, J. C. N. RUP como Metodologia de Desenvolvimento de Software para Obtenção da Qualidade de Software. In: **SEGeT** – Simpósio de Excelência em Gestão e Tecnologia. Resende: Associação Educacional Dom Bosco, 2008.

## REFERÊNCIAS BIBLIOGRÁFICAS

_____. **Defining Business Rules** - What Are They Really. Disponível em: http://www.businessrulesgroup.org/first_paper/br01c0.htm. Acesso em: 28/08/2016.

_____. IBM. Acesso em: http://www.ibm.com/developerworks/downloads/r/rup/?S_CMP=rnav. Acesso em: 31/08/2016.

BARROCA, L., GIMENES, I. M. S., HUZITA, E.H.M. **Desenvolvimento Baseado em Componentes**. Rio de Janeiro: Ciência Moderna, 2005.

GAMMA, E., et al. **Padrões de Projeto:** soluções reutilizáveis de software Orientado a Objetos. Porto Alegre: Bookman, 2000.

GIMENES, I. M. S., HUZITA E. H. M. **Desenvolvimento Baseado em Componentes**. Rio de Janeiro: Ciência Moderna, 2005.

KROLL, P., KRUCHTEN P. **The rational unified process made easy:** a practitioner's guide to the RUP. Boston: Addison Wesley, 2003.

KRUCHTEN, P. **The rational unified process:** an introduction. Boston: Addison-Wesley , 2003.

LEITE, J.C.S.P., LEONARDI, M.C. Business Rules as organizational policies. In: **Proceedings of the 9th International Workshop on Software Specification & Design**. Los Alamitos: IEEE Computer Society, 1998.

MACORATTI, J. C. **Padrões de projeto:** O modelo MVC - Model View Controller. Disponível em: http://www.macoratti.net/vbn_mvc.htm. Acesso em: 30/03/2016.

MARTINS, J. C. C. **Gerenciando projetos de desenvolvimento de software com PMI, RUP e UML**. Rio de Janeiro: Brasport, 2007.

NASCIMENTO, L. B. **Componentização de Software em JAVA2 Micro Edition** – Um framework para Desenvolvimento de Interface Gráfica para Dispositivos Móveis. Recife: Universidade Federal de Pernambuco, 2005.

PAGANO, V. A. **Uma Abordagem Arquitetural com Tratamento de Exceções para Sistemas de Software Baseado em Componentes**. Campinas: Universidade Estadual de Campinas, 2004.

REENSKAUG, T. M. H. **MVC XEROX PARC 1978 - 79**. Disponível em: http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html. Acesso em: 30/08/2016.

RIBEIRO, R. T. **MVC:** a essência e a web. Disponível em: http://rubsphp.blogspot.com.br/2013/02/mvc-essencia-e-web.html. Acesso em: 30/08/2016.

ROSS, R. G. **Business Rules Concepts**. Houston: Business Rules Solutions Inc., 1998.

SANTOS, I., et al. **Possibilidades e limitações da arquitetura mvc (model – view – controller) com ferramenta ide (integrated development environment)**. Alfenas: Universidade José do Rosário Vellano, 2010.

SOMMERVILLE, I. **Engenharia de Software**. São Paulo: Addison Wesley, 2003.

ZEMEL, T. **MVC (Model – View – Controller)**. Disponível em: <http://codeigniterbrasil.com/passos-iniciais/mvc-model-view-controller/>. Acesso em: 30/08/2016.

# 5

# Provisionamento e construção

# Provisionamento e construção

## Introdução

No contexto de arquitetura de sistemas, é importante conhecermos como funciona a arquitetura voltada para sistemas distribuídos e como funciona a comunicação entre esses sistemas remotos que estão integrados entre si em ambiente heterogêneo.

Porém, desenvolver sistemas distribuídos não é uma tarefa fácil justamente por envolver uma série de objetos que estão separados remotamente entre servidores, que, no nosso caso, são os componentes de *softwares* que estudamos em capítulos anteriores. E, neste capítulo, conheceremos o *Framework* CCM que nos ajudará a entender melhor tal funcionamento.

Diante disso, neste capítulo estudaremos pontos importantes a fim de entendermos a importância de aplicar *framework* no desenvolvimento de sistemas distribuídos bem como ade aprender os seus principais conceitos, arquitetura e a importância da interoperabilidade na comunicação entre componentes.

## OBJETIVOS

- Compreender o *Framework* CCM;
- Entender a Arquitetura CORBA;
- Adquirir uma visão geral acerca do CORBA;
- Entender a fase de empacotamento e de distribuição de componentes;
- Compreender a importância dos componentes CORBA;
- Compreender a importância da Arquitetura CORBA na integração de sistemas heterogêneos.

## Framework CCM - Corba Component Model

A OMG (*Object Management Group*) desenvolveu o CORBA (*Common Object Request Broker Architecture*) com o **intuito de agilizar a comunicação entre objetos distribuídos dentro de um mesmo ambiente heterogêneo**, que é o sistema. E, para tal, ele é formado por uma série de especificações que não apresentam restrições para determinado objeto distribuído (OMG, 2016,

VINOSKI, 1999). **A arquitetura do CORBA é composta por um ORB** (*Object Request Broker*), e esse possui a **responsabilidade de gerenciar, de maneira transparente, as comunicações, além de tratar das facilidades no que envolve os elementos do sistema e os de serviços**, que são os elementos direcionados ao usuário. Logicamente, como desenvolvedor do CORBA, o OMG disponibiliza essas especificações que são tratadas como uma série de padrões de facilidades e de serviços.

## A arquitetura

Na figura a seguir, é apresentada a arquitetura do CORBA, o seu modelo de referência. E, logo após, há uma breve descrição de seus conceitos essenciais para que possamos compreender como a comunicação é implementada.

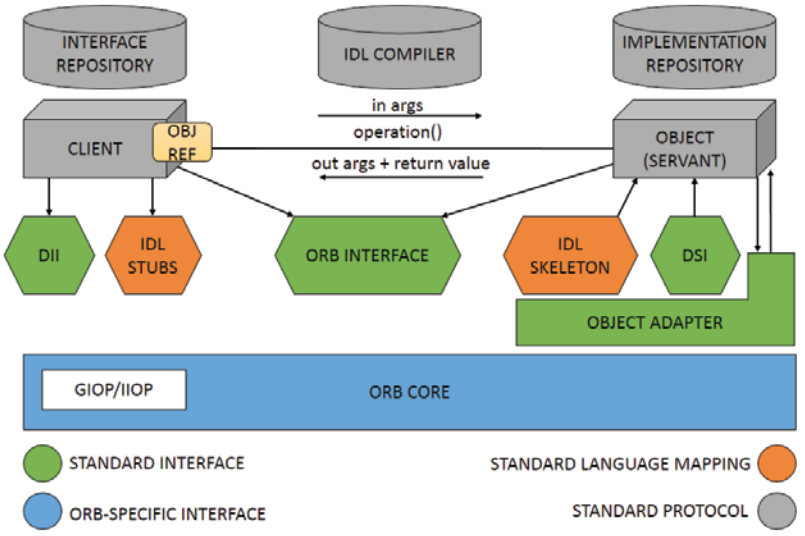


Figura 1. Modelo de referência do CORBA. Adaptado de (OMG, 2016).

- ***Object Implementation:*** responsável por definir operações que são descritas pela interface IDL CORBA. A sua implementação pode ser realizada por utilização de várias linguagens de programação;
- ***Client:*** trata-se do lado Cliente que realiza uma solicitação de serviço ao Servidor. Esse acesso aos serviços remotos deve ser transparente. E, por transparente, entende-se que a sintaxe do código de comunicação não está encapsulada;

• ***Object Request Broker:*** um mecanismo é disponibilizado pelo ORB com o objetivo de fornecer a transparência de comunicação entre cliente e servidor. Dessa forma, a programação distribuída é simplificada devido ao fato de ela intermediar as chamadas de serviços. O cliente solicita o serviço ao servidor, e o ORB tem a responsabilidade de encontrar esse servidor, de executar a solicitação e gerar um retorno como resposta ao cliente quando for necessário;

• ***Interface* ORB:** no CORBA, essa interface é definida como abstrata, o que faz com que ela possua uma série de operações genéricas. Isso permite que a *Interface* ORB seja codificada de diversas formas;

• **CORBA IDL *stubs e skeletons*:** esses são elementos que buscam a união entre cliente e servidor por intermédio do ORB. Faz-se necessária a utilização de compilador CORBA IDL específico quando se quer transformar as definições do CORBA IDL para alguma linguagem de programação, que geralmente é a do sistema;

• ***Dynamic Invocation Interface (DII):*** com essa interface, é permitido ao cliente o acesso direto ao mecanismo responsável por realizar as chamadas de mensagens fornecidas pelo ORB sem que seja necessária a utilização de algum IDL *Stub*;

• ***Dynamic Skeleton Interface (DSI):*** essa interface é análoga à DII que pertence ao cliente. A DSI é a versão de servidor, e é ela quem permitirá que as requisições sejam entregues ao servidor por meio do ORB; assim, nesse momento, o servidor não possui o conhecimento sobre o tipo de objeto que estará sendo implementado por ele. O cliente não possui o conhecimento de que irá se comunicar com uma DSI ou com uma IDL *Skeleton* ao realizar chamada;

• ***Object Adapter:*** ele atua em conjunto com o ORB no servidor com o intuito de auxiliá-lo na inicialização do servidor e com a entrega de requisições de serviços.

Tabela 1. Conceitos essenciais do modelo de referência da arquitetura CORBA.

| ELEMENTO | DESCRIÇÃO |
|---|---|
| *OBJECT IMPLEMENTATION* | Define operações que são descritas pela interface IDL CORBA. |
| *CLIENT* | Realiza solicitação de serviço ao Servidor. |

| | |
|---|---|
| *OBJECT REQUEST BROKER* | Responsável por encontrar o servidor solicitado, executar a solicitação e gerar um retorno como resposta ao cliente quando for necessário. Fornece transparência na comunicação. |
| *INTERFACE* ORB | Possui uma série de operações genéricas. |
| CORBA IDL *STUBS E SKELETONS* | Integram cliente e servidor por intermédio do ORB. |
| *DYNAMIC INVOCATION INTERFACE (DII)* | Permite ao cliente o acesso direto ao mecanismo responsável por realizar as chamadas de mensagens fornecidas pelo ORB. |
| *DYNAMIC SKELETON INTERFACE (DSI)* | Permite que as requisições sejam entregues ao servidor por meio do ORB. |
| *OBJECT ADAPTER* | Ajuda na inicialização do servidor e na entrega de requisições de serviços. |

O OMG padronizou esse processo em dois níveis. No primeiro, o **tipo de objeto é de conhecimento do cliente (é chamado pelo *stub*) e do servidor (é chamado pelo *skeleton*), já que eles são gerados na mesma IDL**. Dessa forma, é de conhecimento do cliente quais são os serviços que ele pode executar, os parâmetros que devem ser utilizados como entrada e o local onde se deve encontrar o serviço desejado. Já no segundo nível, **através do mesmo protocolo é que o ORB cliente e o ORB servidor se comunicam**. Embora o IIOP (*Internet Inter* ORB *Protocol*) seja disponibilizado pela OMG, existe a possibilidade de utilizar outros protocolos desde que eles sigam o modelo genérico denominado GIOP (*General Inter* ORB *Protocol*).

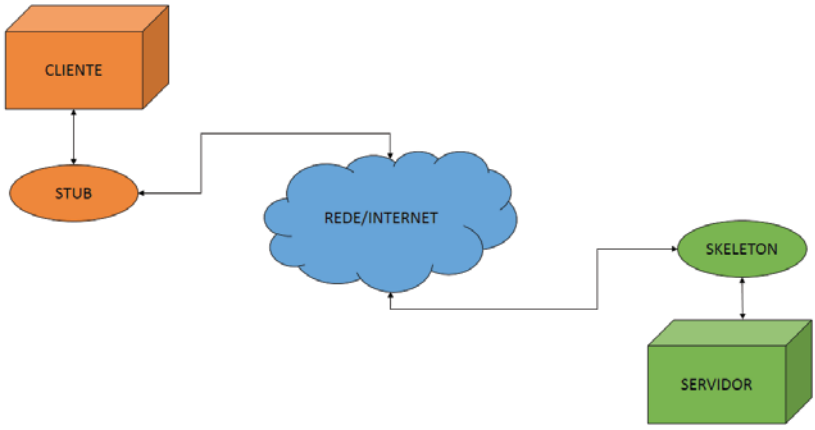


Figura 2. Utilização de stub e skeleton na comunicação cliente/servidor.

## Visão geral

O CCM estende a IDL (*Interface Definition Language*) para que ela possibilite a descrição de componentes CORBA e incorpora o CIDL (*Component Implementation Definition Language*) para a descrição de detalhes acerca da implementação com o objetivo de que seja permitido que o servidor possa gerar, empacotar e implantar toda a estrutura. O mapeamento realizado com o *Enterprise JavaBeans* é definido pelo CCM, permitindo assim que um EJB possa ser visto como um componente CORBA. A maior parte do que é especificado do CCM é baseada nas experiências que os desenvolvedores possuem com modelos de componentes COM+ e EJB. E isso também foi baseado nas experiências de desenvolvimento de estruturas e aplicações para a construção de servidor CORBA. As partes que interagem entre si de forma a se integrarem e relacionarem, compondo o CCM, são:

- **Modelo de Componentes Abstratos** – extensões do modelo de objetos CORBA e IDL;
- **CIF** (*Component Implementation Framework*) – CIDL;
- **Modelo de Programação de Contêineres** – descrição do ambiente de desenvolvimento do lado servidor e apresenta a visão do lado cliente;
- **Arquitetura de Contêineres**;
- **Integração com Eventos, Persistências e Transações**;
- **Integração com EJB**.

No CORBA, a IDL especifica os componentes e o Repositório de Interfaces que podem os representar. A sua implementação e representação interna devem ser encapsuladas pelo componente, procurando apresentar aos clientes quais são as suas operações para que eles, ou até mesmo outros serviços CORBA e elementos da aplicação, possam interagir por meio de portas (ports).

```
interface X, T;
component Exemplo supports X, T          // interface equivalente Exemplo definida
{                                        // interface X e T definidas aceitas por herança
        provides A, B, C, D;             // Facetas
        ...                              // outras definições que o componente pode possuir
}
```

Figura 3. Representação de Componente em IDL.

Na figura acima, temos um componente Exemplo que é instanciado: ele é apresentado ao lado cliente como um tradicional objeto CORBA. Dessa forma, **clientes que não possuem o conhecimento sobre componentes podem requisitar serviços por meio de instâncias, identificando de forma única a instância do componente**, que pode, quando necessário, herdá-lo de outras interfaces. Com isso, existe um certo grau de dificuldade para que os objetos CORBA possam ser estendidos por herança. E por isso existem as facetas (*facets*), que são uma alternativa, as quais admitem as interfaces que não se relacionam por meio da composição. O lado cliente pode pecorrer as interfaces de determinado componente através da interface *Navigation*, que é definida e herdada por todos os outros componentes. Dessa forma, no momento em que se tem uma referência de alguma interface que determinado componente disponibiliza, existe a possibilidade de alcançar referência às suas interfaces.

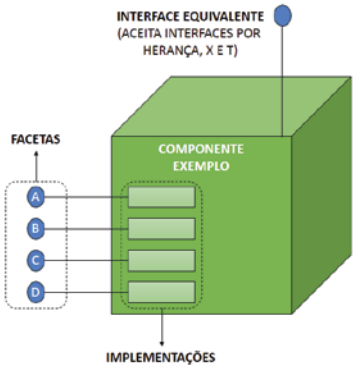


Figura 4. Representação do componente.

O CCM possui quatro tipos de portas. A **faceta** é uma delas, e os outros tipos são:

- **Receptáculos:** permite que determinado componente possa utilizar referências que são disponibilizadas por elementos externos;
- **Fontes de eventos:** publicam eventos para consumidores ou para algum canal de eventos;
- **Consumidores de eventos:** realiza a solicitação de eventos.

O CCM também é composto por outros elementos:

- **Atributos:** auxiliam na configuração de componentes e, assim, permitem funcionalidades de atribuição e acesso;
- **Chaves primárias:** identificam uma instância específica de determinado componente;
- **Interface Home:** gerencia instâncias que fornecem serviços padronizados, além de gerenciar o ciclo de vida relacionado aos componentes e à associação existente entre as instâncias e as chaves primárias.

Para que sistemas possam ser desenvolvidos utilizando o CORBA, alguns passos devem ser seguidos:

1. **Escrita de especificação IDL**;
2. **Compilação dessa especificação** com a ajuda de um compilador IDL;
3. **Escrita de implementação para interface** gerada no item anterior;
4. **Escrita da classe principal**, que será responsável por instanciar e registrar a implementação do lado servidor no ORB;
5. **Escrita da classe cliente**, para que seja possível recuperar determinada referência de objeto no lado servidor através de solicitações remotas.

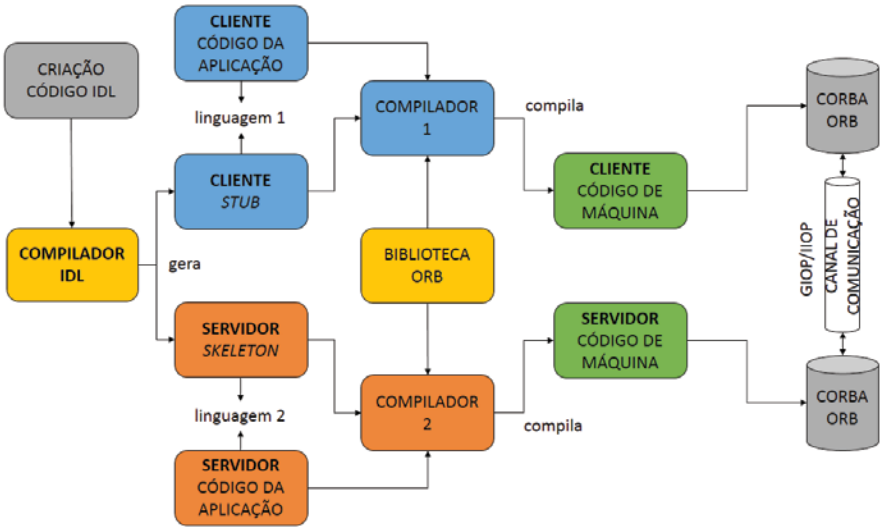


Figura 5. Visão geral da interação entre componentes do CORBA.

## Arquivos CIF e CIDL

Um vasto número de interfaces são definidas pelo CCM com o intuito de que elas comportem funcionalidades dos componentes e, principalmente, a estrutura deles. E muitas dessas interfaces possuem a possibilidade de serem geradas de forma automática. E é aí que entra o CIF. **O CIF utiliza o CIDL para que os *skeletons* possam produzir de forma automática algumas tarefas básicas**.

A seguir, é apresentado o funcionamento do desenvolvimento/compilação de CCM utilizando CIF.

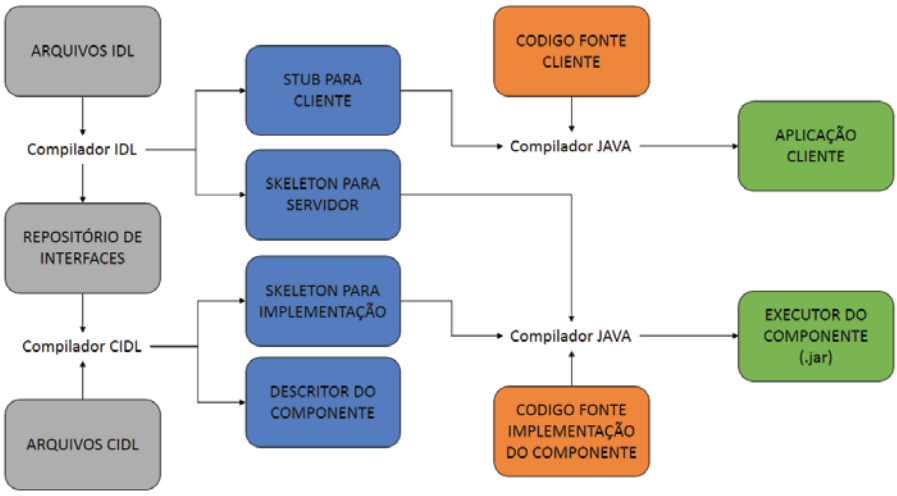


Figura 6. CCM Implementation Framework.

Implementar componente CORBA é uma tarefa que envolve uma série de artefatos que possuem a responsabilidade de apresentar quais são os relacionamentos e comportamentos para que seja construído um componente funcional e completo. Nesse contexto, dois **tipos de artefatos devem estar incluídos: os gerados automaticamente e os implementados pelo programador**. O primeiro disponibiliza toda uma infraestrutura básica do componente, como, por exemplo, *skeletons*, ativação, gerência de ciclo de vida, consultas à identificação etc. Já o segundo trata-se do comportamento do componente que está relacionado ao negócio, representando, dessa forma, uma menor parte entre os artefatos. Toda essa série ou conjunto de artefatos, bem como a sua definição, é chamada de composição.

## Contêineres

Os contêineres foram desenvolvidos com o **objetivo de integrar determinado componente aos serviços CORBA em tempo real**, como, por exemplo, transações, notificações, persistências e segurança. Algumas de suas operações são:

- Ativar e desativar implementações de componentes, preservando ao mesmo tempo recursos de sistema;

• Fornecer camada de adaptação com serviços de transação, persistência, segurança e notificação;
• Fornecer camada de adaptação para *callbacks*;
• Gerenciar políticas do POA (*Portable Object Adapter*).

A arquitetura do modelo de programação de contêineres é responsável por descrever a relação existente entre contêiner e os outros elementos CCM (ver Figura 7).

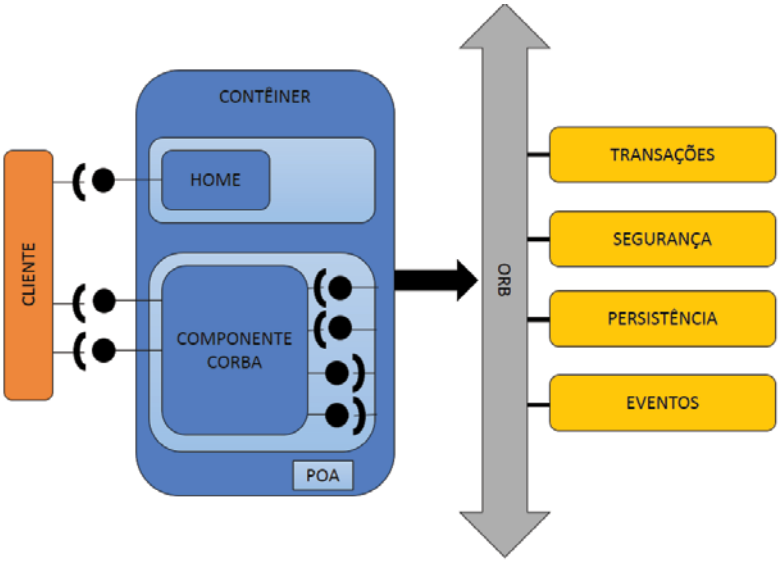


Figura 7. Arquitetura do modelo de programação de contêiners. Adaptado de (CCM, 2002).

Os elementos que fazem parte desse modelo são:

• **APIs externas:** auxiliam a comunicação cliente-componente, correspondem às interfaces que são disponibilizadas para os clientes, são definidas em IDL e são guardadas no repositório de interfaces;
• **APIs do contêiner:** auxiliam a comunicação contêiner-componente, são formadas por *callback* e interfaces internas;
• **Modelo de utilização CORBA:** especifica a integração entre POA e serviços CORBA;
• **Categorias de componentes:** trata-se de uma série de combinações dos três elementos já citados e possui sua especificação em formato de arquivo CIDL.

Quando tratamos de empacotamento envolvendo sistemas desenvolvidos com componentes ou sistemas componetizados, estamos tratando de gerar uma série de arquivos que podem ser classificados em:

• **Arquivos cujo conteúdo está relacionado com a implementação dos componentes**, como, por exemplo, arquivos no formato .class na linguagem Java;

• **Arquivos chamados de descritores**, que basicamente são do formato XML, gerados de forma automática pelo compilador CIDL ou ainda por alguma outra aplicação utilizada para o empacotamento;

• **Arquivos de pacote**, contendo os arquivos anteriores comprimido em formato ZIP.

A figura a seguir ilustra bem o esquema genérico do empacotamento de componentes CORBA.

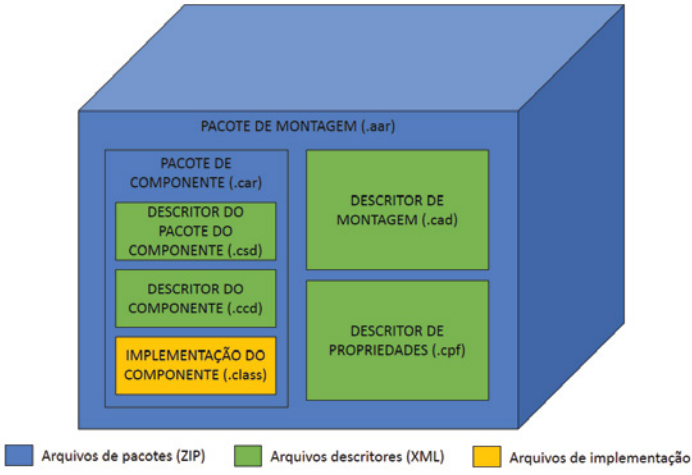


Figura 8. Empacotamento de componentes CORBA.

• **Arquivos descritores:** embora sejam gerados de forma automática, esses tipos de arquivo podem sofrer alterações feitas pelo usuário de forma manual. No CORBA, eles podem ser:

✓ **Arquivo descritor de pacote do componente (.csd):** cada componente possui o seu arquivo descritor de pacote, onde se encontram todos os detalhes acerca da implementação, bem como os sistemas operacionais e o processador que suporta a aplicação, linguagem de programação e compilador. Além dessas informações, é possível encontrar outras, como a localização de arquivos de propriedades, descritor do componente e, ainda, a porta de entrada do componente;

✓ **Arquivo descritor do componente (.ccd):** possui informações acerca da implantação do componente no contêiner. Cada componente deve possuir o seu arquivo descritor, que contém informações sobre serviços, identificação, interfaces, facetas, receptáculos, categoria e demais características do componente;

✓ **Arquivo descritor de montagem (.cad):** cada sistema deve possuir apenas um arquivo descritor de montagem, pois é ele que contém todas as informações que são necessárias para que os componentes possam ser implantados. Tais informações são: localização do pacote, nome e localização do *home* e de cada um dos componentes utilizados no sistema, nome das instâncias, arquivos descritores de propriedades, conexões de interfaces etc.;

✓ **Arquivo descritor de propriedades (.cpf):** nele estão contidos valores sobre os atributos que fazem parte de cada instância do componente.

• **Arquivos de pacotes:** entre os arquivos de pacotes CORBA, podemos encontrar:

✓ **Pacote de montagem (.aar):** cada componente possui o seu pacote de montagem composto por todos os arquivos de pacote de componente ou ainda pelo próprio arquivo de pacote de componente (.car), além dos arquivos de montagem e descritores de propriedades;

✓ **Pacote de componente (.csd):** nesse pacote, encontram-se todos os arquivos que possuem alguma relação com implementação de algum componente, descritores do componente ou, ainda, com o pacote de componente. Esse arquivo de pacote não possui obrigatoriedade, porém a sua existência está relacionada a uma melhor organização de arquivos relacionados com a implantação.

Logo após a geração de todos esses arquivos já citados, eles serão distribuídos para todos os equipamentos onde o sistema será implantado. E, bem logo após

essa distribuição, é realizado o início do processo chamado implantação. O processo de implantação utilizando o CCM deve ser o mais simples possível e pode ser resumido da seguinte forma:

• A aplicação responsável da implantação deve solicitar ao usuário as informações necessárias acerca dos locais onde o sistema será instalado. Ao receber tais informações, a aplicação as armazena e uma cópia segue para o arquivo de pacote de montagem;

• A aplicação responsável da implantação trata de instalar o sistema de componentes seguindo as informações coletadas;

• Uma instância de componentes é criada para que o contêiner seja criado, e dentro dele o *home* do componente será instalado;

• Para que a instância relacionada a cada componente possa ser criada, é utilizado o *home*;

• Cada componente receberá um configurador sempre que necessário;

• Após a instalação e configuração dos componentes, as conexões entre eles são estabelecidas;

• Para cada um dos componentes, a operação *configuration_complete* é chamada.

## Componentes

Dentro da especificação do CORBA, **um componente pode assumir uma classificação entre os tipos básico ou estendido**. O componente *home* gerencia essas duas classificações, porém os serviços disponibilizados são bem diferentes. **No componente básico, os objetos CORBA podem ser componentizados**. Em termos de operações, com a utilização de EJB é realizada uma contribuição significativa para que as soluções possam entregar ambos os tipos. No entanto, **o tipo básico não possui a capacidade de fornecer os quatro tipos de portas** (facetas, receptáculos, consumidores e fontes de eventos). O tipo básico **fornece somente atributos e interface equivalente. Já o componente estendido trata-se de uma série ampliada de operações que são disponibilizadas bem como o tipo de porta**.

Para que componentes sejam definidos, tal definição, no CORBA, ocorre através de declaração na IDL. Essa declaração se divide em: cabeçalho, com o

uso da palavra-chave *component*; e corpo, com o uso de conteúdo entre chaves. Ao ser compilada, essa definição gera uma interface que possuirá as operações que foram definidas dentro do corpo da definição. Essa interface que é gerada estende a definição de interfaces do CORBA, no qual é dado suporte para as novas operações disponibilizadas pelo CCM, como portas e atributos.

No CCM, não é especificada uma palavra-chave definida para que haja uma diferença entre declarar um componente básico ou estendido. Mas todo componente do tipo básico é declarado em IDL da seguinte forma:

"component" <identifier> [<supported_interface_spec>]
"{" {<attr_dcl> ";"}* "}"

Os componentes estendidos são declarados com elementos adicionais, como, por exemplo, herança ou facetas. Seguem alguns exemplos de declarações de componentes:

- **Declaração simples:**
  - ✓ IDL3:
  componentcomponent_name { ... };
  - ✓ IDL2 (interface equivalente):
  interfacecomponent_name: Components::CCMObject { ... };

- **Interfaces admitidas por heranças:**
  - ✓ IDL3:
  component<component_name>
  supports<interface_name_1>, <interface_name_2> { ... };
  - ✓ IDL2 (interface equivalente):
  interface<component_name>: Components::CCMObject
  <interface_name_1>, <interface_name_2>{ ... };

- **Herança de componente:**
  - ✓ IDL3:
  component<component_name> : <base_name> { ... };
  - ✓ IDL2 (interface equivalente):
  interface<component_name>: <base_name> { ... };

- **Herança de componente e interfaces admitidas por herança:**
  - ✓ IDL3:
    component<component_name> : <base_name>
    supports<interface_name_1>, <interface_name_2> { ... };
  - ✓ IDL2 (interface equivalente):
    interface<component_name>: <base_name>,
    <interface_name_1>, <interface_name_2> { ... };

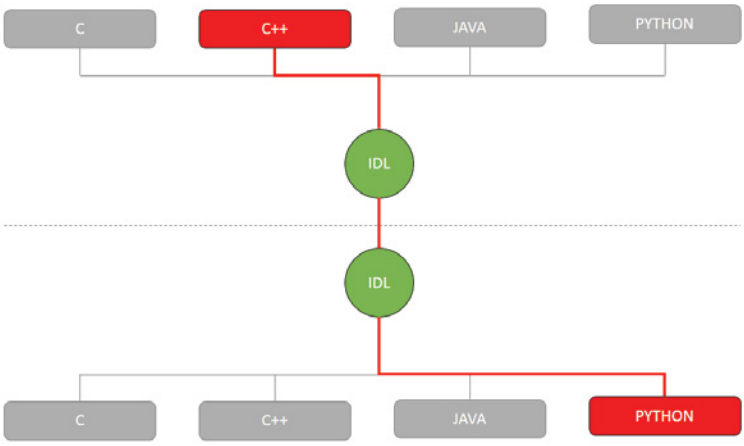


Figura 9. Uso de IDL entre duas linguagens de programação.

### Integração com sistemas existentes

É comum encontrar sistemas com diferentes características, seja por questões de *hardware* ou *software*; nesse contexto, estamos tratando de heterogeneidade. O mesmo ocorre com os diferentes componentes que podem compor um sistema, em que cada um possui a própria configuração e modo de se comunicar com o ambiente da aplicação. A necessidade de integrar sistemas é um fato, principalmente no que envolve a questão e os modelos de arquiteturas distribuídas. **Vários são os modelos desenvolvidos que buscam auxiliar e atender às características da distribuição, como a disponibilidade, o desempenho, a concorrência, otimização de custos, a falta de estado global, o afastamento, evolução, mobilidade, assincronismo e, principalmente, a heterogeneidade** (ISO 10746-1, 93).

Como já foi tratado neste capítulo, o CORBA foi desenvolvido a fim de permitir o tratamento e desenvolvimento de sistemas heterogêneos por meio de objetos com o intuito de integrar diferentes sistemas. Na figura a seguir, é mostrada uma solicitação de serviço sendo solicitada através do ORB do CORBA; nela, podemos observar que o cliente não possui dependência em relação à linguagem de programação à qual o objeto foi codificado nem em relação à localização do objeto, e muito menos por outro fator que não esteja presente na interface do objeto.

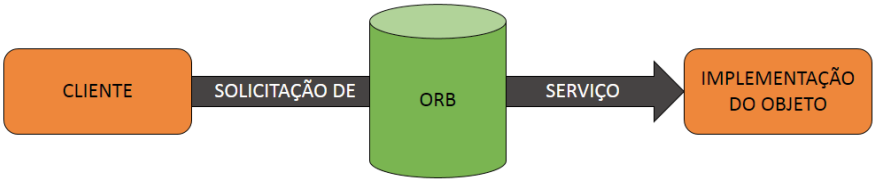


Figura 10. Serviço solicitado por meio do ORB.

A solicitação de serviço realizada pelo cliente e a definição de interfaces do objeto podem ser das seguintes formas:

- **Estática:** para que uma interface do objeto seja definida dessa forma, é necessário utilizar a IDL, que possibilitará a definir de acordo com seu tipo de operação e os parâmetros que serão necessários. Do lado cliente, é gerado uma IDL *stub* para que seja permitido ao cliente acessar o objeto através do ORB. E, na codificação, o objeto deve possuir um IDL *skeleton*;
- **Dinâmica:** de forma dinâmica, determinada interface poderá ser adicionada a algum serviço de Repositório de Interfaces. Nesse serviço, são representados os componentes de interface única, a exemplo de objetos. Sendo assim, o acesso a esses componentes, durante a execução, é permitido. A DII é utilizada para que o cliente tenha acesso ao objeto definido de forma dinâmica.

Dessa forma, **a codificação do objeto aceita determinada solicitação como sendo do ORB por meio de *skeleton* gerado de uma IDL ou ainda de um *skeleton* dinâmico**. O poder de expressão de IDL e o de Repositório de Interfaces, seja em qualquer codificação de ORB, serão equivalentes.

**ORBs são fornecidos por vários fornecedores: a utilização do padrão CORBA os permite serem interoperáveis, seguindo a OMG IIOP ou GIOP**. Esse protocolo permite o envio de solicitações de serviços para objetos distribuídos que são gerenciados em outros domínios por ORBs. A figura a serguir apresenta a utilização do IIOP na comunicação entre dois ORBs:

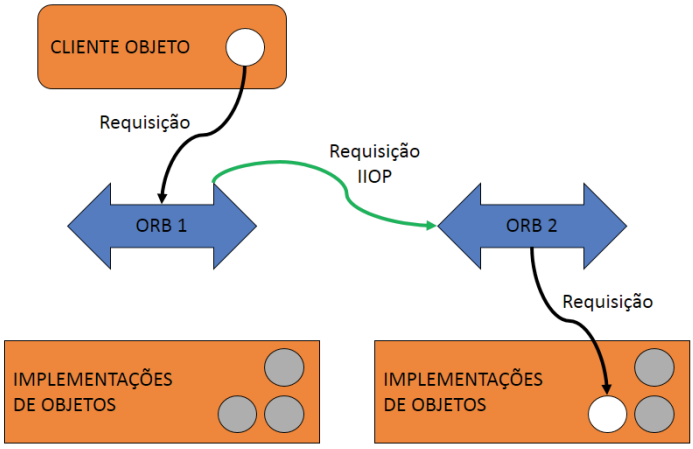


Figura 11. Comunicação entre dois ORBs utilizando o IIOP.

Dessa forma, **os objetos que se encontram distribuídos podem, através da utilização do CORBA, estar presentes em qualquer lugar da rede** (ORFALI, 1996). Tais objetos são empacotados em formato de componentes binários, os quais podem ser acessados por clientes através dos métodos que são invocados. E existe transparência em relação à linguagem e à compilação que são utilizadas no lado servidor para a criação desses objetos. Alguns de seus principais pontos positivos:

- **Acesso de dados de forma transparente**, em que o uso do ORB possibilita o acesso aos objetos sem depender de sua localização;
- **A não dependência de linguagem de programação:** aqui existe a utilização do padrão IDL no qual as interfaces são, em sua totalidade, definidas;
- **Plataforma neutra**, pois o IIOP permite a comunicação entre diversos ORBs.

De fato, o padrão estudado neste capítulo garante, sim, a integração de sistemas em ambientes caracterizados como heterogêneos. Porém o CORBA não

somente auxilia na comunicação entre objetos distribuídos, mas também **fornece infraestrutura em relação a utilitários e serviços; por meio de ORB, ele garante o acesso a objetos graças às solicitações de serviços sem ter a necessidade de saber a localização desses objetos, possibilitando, assim, a interoperabilidade entre diferentes ORBs, além de, claro, não haver a dependência de linguagem graças à utilização de IDL.**

## ATIVIDADE

**01.** De acordo com o que foi estudado neste capítulo, o que é o CORBA?

**02.** De acordo com o que foi estudado neste capítulo, o que é o ORB?

**03.** De acordo com o que foi estudado neste capítulo, descreva os conceitos essenciais da arquitetura CORBA.

**04.** De acordo com o que foi estudado neste capítulo, descreva os tipos de portas de componentes.

**05.** De acordo com o que foi estudado neste capítulo, descreva as classificações dos arquivos que são gerados após o empacotamento de componentes CORBA.

**06.** De acordo com o que foi estudado neste capítulo, descreva de forma simples e resumida o processo de implantação utilizando o CCM.

## GABARITO

**01.** O CORBA (*Common Object Request Broker Architecture*) é uma arquitetura que possui o objetivo de agilizar a comunicação entre objetos distribuídos dentro de um mesmo ambiente heterogêneo. E, para isso, ele é composto por uma série de especificações que não apresentam restrições para determinado objeto distribuído.

**02.** O ORB (*Object Request Broker*) possui a responsabilidade de gerenciar, de maneira transparente, as comunicações, além de tratar das facilidades no que envolver os elementos do sistema e os de serviços, que são os elementos direcionados ao usuário.

**03.** ▪ *Object Implementation:* responsável por definir operações que são descritas pela interface IDL CORBA. A sua implementação pode ser realizada por utilização de várias linguagens de programação;

▪ *Client:* trata-se do lado Cliente que realiza uma solicitação de serviço ao Servidor. Esse acesso aos serviços remotos deve ser transparente. E, por transparente, entenda-se que a sintaxe do código de comunicação não está encapsulada;

▪ *Object Request Broker:* um mecanismo é disponibilizado pelo ORB com o objetivo de fornecer a transparência de comunicação entre cliente e servidor. Dessa forma, a programação distribuída é simplificada devido à intermediação das chamadas de serviços. O cliente solicita o serviço ao servidor, o ORB tem a responsabilidade de encontrar esse servidor, de executar a solicitação e gerar um retorno como resposta ao cliente quando for necessário;

▪ *Interface ORB:* no CORBA, essa interface é definida como abstrata, o que faz com que ela possua uma série de operações genéricas. Isso permite que a Interface ORB seja codificada de diversas formas;

▪ *CORBA IDL stubs e skeletons:* esses são elementos que buscam a união entre cliente e servidor por intermédio do ORB. Faz-se necessária a utilização de compilador CORBA IDL específico quando se quer transformar as definições do CORBA IDL para alguma linguagem de programação, que geralmente é a do sistema;

▪ *Dynamic Invocation Interface (DII):* com essa interface, é permitido ao cliente o acesso direto ao mecanismo responsável por realizar chamadas de mensagens fornecidas pelo ORB sem que seja necessária a utilização de algum IDL *stub*;

▪ *Dynamic Skeleton Interface (DSI):* essa interface é análoga à DII que pertence ao cliente. A DSI é a versão de servidor, e é ela quem permitirá que as requisições sejam entregues ao servidor por meio do ORB; e, nesse momento, o servidor não possui o conhecimento sobre o tipo de objeto que estará sendo implementado por ele. O cliente não possui o conhecimento se ele irá se comunicar com uma DSI ou com uma IDL *Skeleton* ao realizar a chamada;

▪ *Object Adapter:* ele atua em conjunto com o ORB no servidor com o intuito de auxiliá-lo na inicialização do servidor e com a entrega de requisições de serviços.

**04.** ▪ Facetas: admitem as interfaces, que não se relacionam, por meio da composição;

▪ Receptáculos: permitem que determinado componente possa utilizar referências que são disponibilizadas por elementos externos;

▪ Fontes de eventos: publicam eventos para consumidores de eventos ou para algum canal de eventos;

▪ Consumidores de eventos: realizam a solicitação de eventos.

**05.** ▪ Arquivos cujo conteúdo está relacionado com a implementação dos componentes, como, por exemplo, arquivos no formato .class na linguagem Java;
▪ Arquivos chamados de descritores, que basicamente são do formato XML, gerados de forma automática pelo compilador CIDL ou, ainda, por alguma outra aplicação utilizada para o empacotamento;
▪ Arquivos de pacote, que contêm os arquivos anteriores comprimidos em formato ZIP.

**06.** ▪ A aplicação responsável da implantação deve solicitar ao usuário as informações necessárias acerca dos locais onde o sistema será instalado. Ao receber tais informações, a aplicação as armazena e uma cópia segue para o arquivo de pacote de montagem;
▪ A aplicação responsável da implantação trata de instalar o sistema de componentes seguindo as informações coletadas;
▪ Uma instância de componentes é criada para que o contêiner seja criado, e dentro dele o home do componente será instalado;
▪ Para que a instância relacionada a cada componente possa ser criada, é utilizado o home;
▪ Cada componente receberá um configurador sempre que necessário;
▪ Após a instalação e a configuração dos componentes, as conexões entre eles são estabelecidas;
▪ Para cada um dos componentes, a operação *configuration_complete* é chamada.

## REFLEXÃO

Neste capítulo, o nosso foco de estudos para o provisionamento e a construção foi feito com a utilização de um *framework* voltado para o desenvolvimento de sistemas distribuídos, o qual se trata de um modelo arquitetural de sistemas que se utiliza da interoperabilidade por meio de uma linguagem definida para facilitar a comunicação e integração de sistemas e componentes de diferentes fornecedores.

Iniciamos os nossos estudos trazendo ao nosso conhecimento o *Framework* CCM (CORBA *Component Model*), que foi desenvolvido pelo OMG para tratar a comunicação entre objetos remotos que fazem parte de um mesmo ambiente/sistema heterogêneo. Aprendemos que a sua arquitetura é composta por ORBs que gerenciam de forma transparente a comunicação entre os objetos remotos. Conhecemos os conceitos que são essenciais na implementação dessa comunicação. Compreendemos também como *stub* e *skeleton* são utilizados na comunicação entre cliente e servidor. Na visão geral do CCM, conhecemos quais são as partes que interagem e compõem o CCM, os tipos de portas presentes nos componentes

CORBA e outros elementos que compõem o CCM. Além dos passos que devem ser seguidos para utilizar o CORBA. E tratamos brevemente sobre IDL.

Estudamos, também, o funcionamento da implementação de CCM utilizando CIF e CIDL. E entendemos que a implementação de componente CORBA envolve um conjunto de artefatos que são responsáveis por apresentar os relacionamentos e comportamentos, construindo um componente funcional e completo. Assim, conhecemos dois tipos de artefatos, os gerados automaticamente e os gerados pelo programador. Sobre os contêineres, conhecemos o seu objetivo de integrar componentes aos serviços CORBA em tempo real e algumas de suas operações, além do seu modelo arquitetural de programação e os elementos que fazem parte desse modelo.

Conhecemos, também, os arquivos que fazem parte do empacotamento de sistemas desenvolvidos com componentes e uma breve descrição de cada um. Verificamos o seu esquema genérico de empacotamento de componentes CORBA e aprendemos os conceitos e os tipos desses arquivos. Entendemos que a distribuição parte da finalização do empacotamento de componentes com a meta de eles chegarem a todos os equipamentos onde o sistema será implantado. E então a implantação é realizada. Conhecemos como a implantação utilizando o CCM é realizada passo a passo.

Concentramos os nossos estudos, também, nos componentes CORBA, em que, através de exemplos com IDL, aprendemos como especificar e definir componentes e interfaces equivalentes com essa linguagem. Compreendemos, também, o funcionamento da comunicação entre objetos distribuídos através de ORBs, o que permite que sistemas possam ser integrados, e entendemos como a interoperabilidade ocorre. E mais uma vez entendemos a importância da IDL na comunicação e integração.

Finalizamos este capítulo com uma base de conhecimentos solidificada acerca de arquitetura de sistemas e de componentes, além da importância da interoperabilidade para a integração de sistemas distribuídos através da utilização da arquitetura CORBA, o uso do *Framework* CCM. Com os conhecimentos adquiridos neste capítulo, finalizamos os nossos estudos acerca de arquitetura de sistema de forma satisfatória, pois obtivemos um entendimento amplo sobre o assunto. Porém devemos estar sempre em constante estudo e atualização, já que sempre algo novo surge ou conceitos são melhorados, principalmente em termos de tecnologias.

## LEITURA

Para você melhorar ainda mais o seu nível de aprendizagem envolvendo os conceitos referentes a este capítulo, consulte as sugestões abaixo:

AMARAL, F. C. T. **Um modelo interativo de ensino à distância utilizando Java e CORBA**. Recife: Universidade Federal de Pernambuco, 1999.

BORGES, P., SILVA, T., PATRÍCIO, F., RAMOS, F. Utilização da Arquitectura CORBA num Sistema de Televigilância. In: **Revista do DETUA, v. 2, n. 4.** XXXX: YYY1999.

LUNG, L. C., FRAGA, J. S., PADILHA, R., SOUZA, L. Adaptando as Especificações FT-CORBA para Redes de Larga Escala. In: **19° Simpósio Brasileiro de Redes de Computadores**. Florianópolis: XXXX, 2001.

GOMES, A. T. A. **CORBA para Computação Móvel**. Rio de Janeiro: Pontifícia Universidade Católica do Rio de Janeiro, 2001.

LICHT, F. L. **Tutorial Corba**. Disponível em: http://virtual01.lncc.br/~licht/tutoriais/tutor/Tutorial%20Corba(licht).doc. Acesso em: 16/09/2016.

ANTONIOLLI, P. D., SALLES, J. A. A. Proposta de Sistema de Informações para Gestão de Demanda e de Inventários nas Cadeias de Suprimentos. In: **XXV Encontro Nacional de Engenharia de Produção**. Porto Alegre: XXXX, 2005.

AUGUSTO, C. E. L., FONSECA, E.; MARQUES, L., ROENICK, H., CERQUEIRA, R. F. G., CORRÊA, S. L. **SCS - Sistema de Componentes de Software**. Disponível em: https://webserver2.tecgraf.puc-rio.br/~scorrea/scs/documentation/scsOverview.pdf . Acesso em: 17/09/2016.

## REFERÊNCIAS BIBLIOGRÁFICAS


_____. ISO – Reference Model of Open Distributed Processing – Part 1. In: **Overview**. XXXX: YYY, 1993.

_____. Object Management Group. In: **CORBA Components, v. 3.0**. Disponível em: http://www.omg.org/cgi-bin/doc?formal/02-06-65. Acesso em: 13/09/2016.

_____. **Object Management Group**. Disponível em: http://www.omg.org. Acesso em: 13/09/2016.

ORFALI, R., HARKEY, D., EDWARDS, J. **The Essential Distributed Objects Survival Guide**. New York: John Wiley & Sons, 1996.

VINOSKI, S. **Advanced CORBA Programming with C++**. Massachusetts: Addison-Wesley, 1999.

## ANOTAÇÕES

# ANOTAÇÕES

## ANOTAÇÕES

## ANOTAÇÕES

## ANOTAÇÕES

# ANOTAÇÕES

## ANOTAÇÕES