Componentes CORBA

Alexandre Ricardo Nardi

Dissertação apresentada ao
Instituto de Matemática e Estatística da
Universidade de São Paulo para
obtenção do grau de
Mestre em Ciência da Computação

Área de Concentração: Sistemas Distribuídos
Orientador: Prof. Dr. Francisco C. R. Reverbel

São Paulo, setembro de 2003

Componentes CORBA

Este exemplar corresponde à redação final
da dissertação, devidamente corrigida,
defendida por Alexandre Ricardo Nardi
e aprovada pela comissão julgadora.

São Paulo, 09 de setembro de 2003

Comissão Julgadora:

- Prof. Dr. Francisco Carlos da Rocha Reverbel (orientador) – IME/USP
- Prof. Dr. Fábio Kon – IME/USP
- Profa. Dra. Graça Bressan – EP/USP

Para Ednilza e Pedro

Agradecimentos

Em primeiro lugar, gostaria de agradecer ao Prof. Reverbel, meu orientador, pela atenção e meticulosidade com que teceu observações no transcorrer deste trabalho mas, acima de tudo, pelo profissionalismo com o qual fui tratado, no que se refere ao ritmo que impusemos durante a escrita deste texto.

À minha esposa, Ednilza, pelo carinho, incentivo e paciência, muito obrigado! E agradeço também ao pequeno (por enquanto) Pedro, meu filho, pela inspiração neste último ano.

Um agradecimento especial, que aqui não poderia faltar, vai para a Opus Software, que proporcionou flexibilidade suficiente para que eu pudesse cursar o mestrado. Especialmente agradeço ao Francisco Barguil, ao Edison Kalaf e ao Patrick Theys pelo apoio que recebi nestes anos.

Por compartilharem um pouco de seu conhecimento comigo, agradeço aos professores do IME, e aos colegas por assistirem às minhas palestras e seminários.

Obrigado aos membros da banca examinadora, Prof. Fábio Kon e Profa. Graça Bressan, pelas críticas que certamente contribuíram para melhorar a qualidade desta dissertação.

Aos meus amigos e familiares, agradeço pelo apoio e interesse em me ouvir falar sobre o curso.

Enfim, a todas as pessoas que de alguma forma contribuíram para a realização deste trabalho, seja por me ouvirem, por lerem o texto em suas versões incompletas ou ainda por se interessarem pelo assunto, meu sincero agradecimento.

Resumo

Esta dissertação apresenta o Modelo de Componentes CORBA, que é parte da especificação CORBA 3.0. Esse modelo representa uma das mudanças mais significativas em relação às versões anteriores de CORBA.

O desenvolvimento de sistemas distribuídos é uma tarefa complexa, envolvendo fatores como a organização de conjuntos de objetos que devem ser implantados e executados em servidores distintos. Aqui estes conjuntos são denominados componentes, conceito já existente em outras especificações, como a dos *Enterprise Java Beans*.

O texto da especificação dos Componentes CORBA, apesar de conter exemplos, mostra-se de difícil compreensão para o desenvolvedor de sistemas. Este trabalho se propõe a facilitar a tarefa de entendimento e utilização de Componentes CORBA, procurando ser didático e ilustrativo o suficiente para que diferentes perfis de leitores possam compreender os diversos conceitos apresentados, seja como uma visão geral, ou ainda como auxílio no desenvolvimento de componentes.

Abstract

This thesis presents the CORBA Component Model, which is part of the CORBA 3.0 specification. The model is one of the most significant additions with respect to previous versions of CORBA.

The development of distributed systems is a complex task, involving factors such as the organization of sets of objects that must be deployed and executed in separate servers. Here, these sets are called components, a concept that already exists in other specifications, such as Enterprise Java Beans.

Even though it contains examples, the CORBA Components specification text is not an easy reading for systems developers. Our text intends to facilitate the task of understanding and using CORBA Components, trying to be didactic and illustrative enough so that different profiles of readers are able to understand the many concepts presented, either as an overview, or still as an aid in components development.

Sumário

Índice de Figuras

Índice de Tabelas

Capítulo 1

Introdução

Desde a publicação de sua primeira versão, em 1991, pelo *Object Management Group* (OMG), CORBA tem se tornado padrão e praticamente sinônimo de flexibilidade e versatilidade quando se trata de comunicação entre objetos, independentemente de linguagem de programação, sistema operacional e mesmo protocolo de rede.

Este, entretanto, é um padrão em constante evolução e amadurecimento, e a versão atual do mesmo, CORBA 3.0 [CORBA3], agrega novas funcionalidades que se propõem a resolver situações desconfortáveis para o desenvolvedor e para os responsáveis pela manutenção de aplicações. O Modelo de Componentes CORBA faz parte destas funcionalidades.

1.1 Objetivo deste Trabalho

Por se tratar de um assunto novo, a literatura é relativamente escassa no que diz respeito aos Componentes CORBA. A fonte mais completa sobre o assunto é a especificação [CCM02] disponibilizada pela OMG. Todavia, trata-se de um texto extenso (a primeira versão com que trabalhamos era composta por três volumes, totalizando 852 páginas) cuja leitura não é trivial, uma vez que se destina aos implementadores de ORBs, e não aos desenvolvedores de aplicações. Além disso, uma vez que a especificação é um documento genérico, não se atendo a alguma implementação em particular, os exemplos lá existentes são incompletos, dificultando a visualização e compreensão do desenvolvimento de uma aplicação, desde as definições em IDL até sua execução no contexto de servidores de aplicação.

Por outro lado, as implementações sendo atualmente desenvolvidas também são incompletas, ilustrando alguns conceitos e deixando outros a serem implementados.

O objetivo deste trabalho é prover um documento mais didático do que a especificação, contendo exemplos para ilustrar os conceitos apresentados. É nossa intenção que os exemplos possam ser executados pelo leitor, sempre que possível.

Procuramos atender a leitores com interesses diversos no tema, indo desde aquele que deseja uma visão geral sobre o assunto, até o que pretende desenvolver aplicações componentizadas.

Não pretendemos, contudo, esgotar o assunto, sendo que alguns conceitos aqui presentes são tratados de forma superficial, como a integração EJB/CCM, por exemplo.

1.2 Organização do Texto

Este texto pressupõe que o leitor esteja familiarizado com orientação a objetos e CORBA. Entretanto, o Capítulo 2 apresenta uma revisão de alguns conceitos de CORBA, no intuito de relembrar o leitor que não utilize esta arquitetura com freqüência.

O Capítulo 3 apresenta os principais elementos do Modelo de Componentes CORBA, como alternativa de solução dos problemas existentes nas especificações CORBA 2.x, descritos no capítulo anterior. Ainda neste capítulo é apresentado um exemplo de utilização que será referenciado em diversos pontos do texto.

No Capítulo 4 detalhamos o Modelo de Componentes CORBA, mostrando como utilizar as novas construções nele contidas. Neste e em outros capítulos apresentamos diversas interfaces e suas operações. Estas informações se destinam ao leitor que deseja implementar uma aplicação componentizada.

Os contêineres, que são o ambiente de execução dos componentes, são o assunto do Capítulo 5.

O Capítulo 6 descreve o arcabouço de implementação de componentes, que apresenta um dos principais recursos do Modelo de Componentes: a geração automática de artefatos (classes, em Java).

O empacotamento e a distribuição de componentes são tratados no Capítulo 7, que inclui exemplos de cada tipo de arquivo envolvido.

Finalmente, no Capítulo 8 tecemos algumas considerações como decorrência de nossa experiência com o Modelo de Componentes, apresentando uma visão pessoal sobre a utilização deste.

Como leitura suplementar, o Apêndice A comenta características de outras tecnologias para desenvolvimento de componentes, como *Enterprise Java Beans* e Microsoft.NET.

O Apêndice B mostra as possibilidades de integração do Modelo de Componentes CORBA com *Enterprise Java Beans*.

O Apêndice C, por sua vez, contém a listagem do código-fonte do exemplo descrito no Capítulo 3.

1.3 Convenções Tipográficas

Neste texto foram utilizadas as seguintes convenções tipográficas:

- Termos em inglês[1] são grafados *em itálico*;
- Trechos de código, Java ou IDL, utilizam a fonte `Courier New, tamanho 9`;
- No corpo do texto, referências a termos utilizados em código fonte ou palavras reservadas (IDL ou Java) aparecem com a fonte `Courier New, tamanho 12`.

[1] Alguns termos empregados no texto foram propositalmente deixados em inglês.

Capítulo 2

Conceitos de CORBA 2.x

Antes de iniciarmos a apresentação dos Componentes CORBA, é importante relembrar ao leitor os principais conceitos de CORBA 2.x [CORBA2]. Esta seção não pretende abordar o assunto em profundidade, tarefa realizada com competência por diversos autores, como Michi Henning e Steve Vinoski no livro "*Advanced CORBA Programming with C++*" [HV99], por exemplo.

A Figura 2.1 apresenta uma visão geral da arquitetura CORBA, e inclui alguns de seus principais constituintes, descritos nas próximas seções.

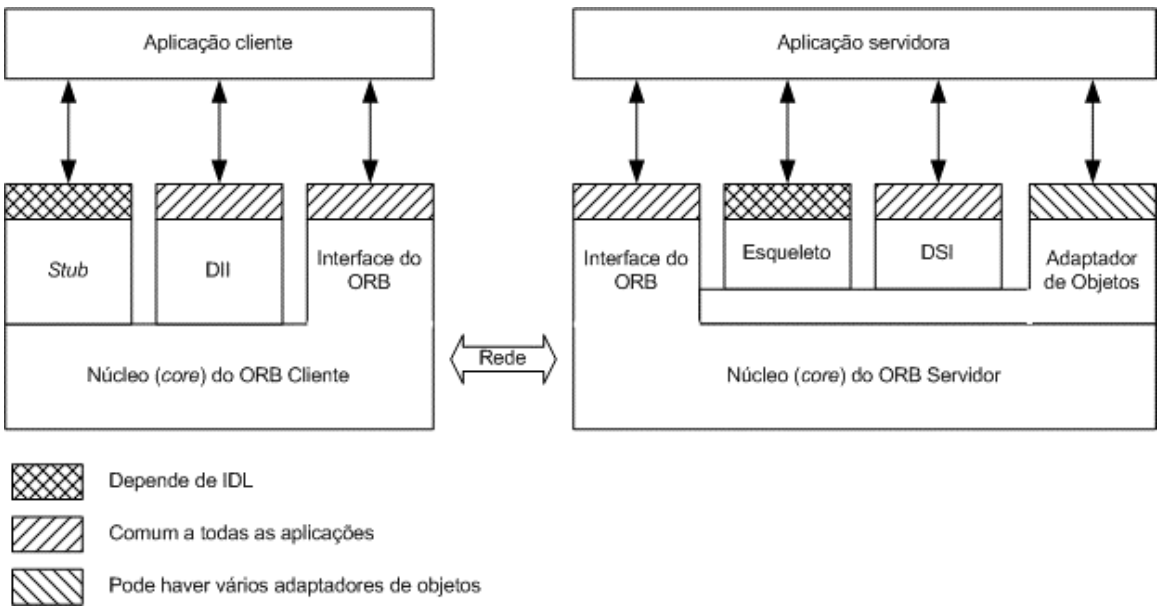

Figura 2.1: visão geral de CORBA

2.1 Terminologia

CORBA possui, do mesmo modo que outras tecnologias, uma terminologia própria. A compreensão dos termos e conceitos empregados é essencial para o entendimento da arquitetura em si. A seguir, citamos e explicamos os termos mais comuns em CORBA:

- Cliente é a aplicação que envia requisições aos objetos CORBA. O cliente pode existir em um espaço de endereçamento diferente do objeto referenciado. Esse fato confere a CORBA sua característica de arquitetura distribuída. O termo cliente é limitado ao escopo de uma requisição, sendo que uma aplicação cliente para uma requisição pode ser um servidor para outra requisição;
- Servidor é a aplicação onde os objetos CORBA residem, recebendo requisições. Assim como para os clientes, o termo servidor limita-se ao escopo de uma requisição;
- Objeto-alvo é aquele, no servidor, a quem se destina uma requisição. É determinado pela referência a objeto utilizada para invocar a requisição;
- Requisição é a invocação de uma operação de um objeto CORBA por um cliente. A requisição flui do cliente para um objeto-alvo no servidor, que envia o resultado ao cliente, se for o caso;
- *Object Request Broker* (ORB) é o elemento de CORBA responsável por encaminhar requisições de clientes aos objetos;
- Objeto CORBA é uma entidade virtual capaz de ser localizada por um ORB e de receber requisições de clientes. O caráter virtual deve-se ao fato do objeto não existir até que uma implementação seja realizada em alguma linguagem de programação;
- Referência a objeto é um *handle* usado para identificar, localizar e acessar um objeto CORBA. Para os clientes essa referência é uma entidade opaca, ou seja, os clientes a utilizam para enviar requisições, mas não podem acessar os constituintes do objeto referenciado. Cada referência diz respeito a um único objeto CORBA;
- Servente é uma entidade que implementa um objeto CORBA em alguma linguagem de programação. Dizemos que os serventes encarnam os objetos CORBA porque lhes fornecem corpo, ou implementações. Em Java, serventes são implementações de uma classe particular, e uma encarnação de um servente é uma instância daquela classe.

2.2 Funcionalidades de CORBA

Nesta seção apresentamos uma visão geral das seguintes funcionalidades de CORBA:

- OMG *Interface Definition Language* [CORBA2];
- Mapeamento para linguagens de programação;
- Invocação de operações;
- Adaptadores de objetos.

2.2.1 OMG Interface Definition Language

Para invocar operações em um objeto CORBA, o cliente deve conhecer a interface oferecida pelo objeto. A interface de um objeto é composta pelas operações que ele oferece, bem como os tipos de dados que podem ser passados de/para essas operações.

Em CORBA, as interfaces são definidas através de uma linguagem específica para esse fim, a *Interface Definition Language* (IDL). Diferentemente das linguagens de programação, a IDL não possui construções como comandos condicionais ou de repetição. Assim, os objetos CORBA não podem ser implementados em IDL, mas apenas definidos de modo independente de qualquer linguagem de programação. Essa característica é ponto chave para a utilização de CORBA em sistemas heterogêneos e integração de aplicações desenvolvidas separadamente.

A IDL fornece suporte a tipos simples, como números inteiros, caracteres, *strings*, e também a tipos complexos, como enumerações, estruturas, e exceções. É possível também a utilização de herança múltipla.

Existe, ainda, um caso especial de herança: todas as interfaces em IDL implicitamente herdam da interface `Object`, definida no módulo `CORBA`. Essa interface define operações comuns a todos os objetos CORBA.

2.2.2 Mapeamento para Linguagens de Programação

Uma vez que as interfaces estejam definidas, estas devem ser compiladas de modo que os tipos de dados e as operações sejam mapeados na linguagem de programação utilizada para o desenvolvimento de servidores e clientes. Existem, para isso, diversos mapeamentos de IDL para linguagens como, por exemplo, C++, Smalltalk, Java e COBOL.

2.2.3 Invocação de Operações

Como resultado da compilação da IDL, são gerados elementos chave de CORBA, chamados *stubs* e esqueletos. Seu objetivo é tornar a invocação de operações independente da linguagem de programação utilizada no desenvolvimento das aplicações clientes e servidoras, nesta ordem.

Assim, se quisermos desenvolver o servidor em C++ e o cliente em Java, por exemplo, basta compilar a IDL em C++ e Java, utilizando o esqueleto gerado na primeira compilação e o *stub* gerado na segunda.

Quando o cliente invocar uma operação do servidor, os seguintes passos, que podem ser observados na Figura 2.1, serão realizados:

1. Conversão de tipos da chamada de Java para IDL (*stub*);
2. Envio da requisição ao servidor;
3. Conversão de tipos de IDL para C++ (esqueleto);
4. Invocação da operação;
5. Conversão do retorno de C++ para IDL (esqueleto);
6. Envio de resposta ao cliente;
7. Conversão do retorno de IDL para Java (*stub*);
8. Retorno à aplicação cliente.

O processo de conversão de tipos durante a invocação de operações é denominado *marshaling*.

Como alternativa ao uso de *stubs* e esqueletos, podem ser utilizadas interfaces de invocação dinâmica (DII) pelo cliente e esqueleto dinâmico (DSI) pelo servidor.

2.2.4 Adaptadores de Objetos

Funcionam como o elo de ligação entre os serventes e os ORBs. Conforme descrito no *design pattern Adapter* [GHJV94], “adaptador de objetos é um objeto que adapta a interface de um objeto a outra interface, esperada pelo chamador”. Em outras palavras, um adaptador de objetos é um objeto que usa delegação para permitir que o cliente invoque requisições em um objeto sem conhecer a verdadeira interface do mesmo.

Os adaptadores de objetos realizam tarefas tais como:

- Criam referências a objetos, que permitem aos clientes acessá-los;
- Garantem que todo objeto alvo seja encarnado por um servente;
- Remetem requisições do ORB no lado do servidor aos serventes encarnando cada um dos objetos alvo.

Até a versão 2.1, CORBA especificava apenas o *Basic Object Adapter* (BOA), que tratava de modo abstrato diversas questões no tocante ao ambiente de execução. Muitos detalhes foram deixados indefinidos, sendo responsabilidade dos implementadores de ORBs resolvê-los. Isso resultou em uma série de problemas de portabilidade [OPE95], resolvidos posteriormente com a adoção do *Portable Object Adapter* (POA) [CORBA2, HV99], introduzido com a versão 2.2 de CORBA.

2.3 Problemas das Especificações CORBA 2.x

O ciclo de desenvolvimento de um sistema envolve fases como análise, modelagem, prototipação, implementação, testes, empacotamento, distribuição e, a partir daí, novas implementações ou modificações, seguidos por novo empacotamento e distribuição, num modelo aparentemente simples, mas que traz complexidades como, por exemplo, a substituição de partes de um sistema por outras que mantenham ou estendam a funcionalidade, tornando o processo enfadonho, suscetível a falhas e pouco voltado ao negócio do cliente.

A especificação de CORBA 2.x [CORBA2] apresenta algumas falhas [WSO00, CCM02], como:

- Falta de um padrão para o empacotamento e a implantação (*packaging and deployment*) dos objetos, o que agrega risco a projetos mais complexos, que utilizem centenas ou milhares de objetos;
- Falta de suporte para o uso de subconjuntos de funcionalidades "comuns". Por exemplo, um grande número de aplicações utiliza apenas um subconjunto das possíveis configurações do POA e, apesar disso, o desenvolvedor deve conhecer muitas políticas do POA para conseguir o efeito desejado. Isso possui o efeito colateral de aumentar a complexidade de modo a comprometer a produtividade do desenvolvedor. A preocupação com tal falta de suporte tem feito com que haja busca por padrões que se repitam com freqüência no desenvolvimento de sistemas, para que se possa agilizar o desenvolvimento e até mesmo utilizar ferramentas de geração de código;
- Dificuldade para estender funcionalidades de objetos CORBA, possível apenas através de herança. Por exemplo, para admitir novas interfaces, o desenvolvedor deve definir nova interface em IDL que herde de todas as interfaces em questão, implementar tal interface e implantar a nova implementação nos servidores;
- Falta de definição de serviços obrigatórios: a aplicação deve possuir código que trate a indisponibilidade de serviços, como no caso do gerenciamento de ciclo-de-vida de objetos: apesar da existência do "*Lifecycle Service*", seu uso não é obrigatório, o que dificulta o trabalho no cliente.

Problemas como esses levam à produção de aplicações com objetos fortemente acoplados (*tightly coupled*), difíceis de modelar, reutilizar, implantar, manter e estender.

Capítulo 3

Visão Geral do Modelo de Componentes CORBA

No sentido de construir aplicações em escala empresarial, os desenvolvedores devem integrar a lógica de negócio em uma arquitetura distribuída que inclui, no mínimo, os serviços de segurança, transações, nome, persistência e eventos. Eles necessitam, ainda, ser capazes de implantar e manter tais aplicações. A flexibilidade que CORBA fornece se apresenta em um conjunto extenso de opções, tornando complexa a configuração das aplicações e dificultando a manutenção e o reaproveitamento das mesmas.

Tal situação culminou na busca por poucos padrões que se repitam em grande quantidade de casos. Com foco no escopo limitado por esses padrões, muito do trabalho tedioso pode ser deixado a cargo de ferramentas de geração de código. Isso se assemelha ao que era feito, em CORBA 2.x, pelo compilador de IDL, que gerava *stubs* e esqueletos para *marshaling* e chamadas remotas de métodos. Com CORBA 3.0 é possível que seja realizada a geração de grande parte do código dos servidores, permitindo que o desenvolvedor concentre sua atenção na lógica de negócio, com a elaboração de componentes.

O arcabouço da *Object Management Architecture* (OMA) para aceitar a definição, geração de código, empacotamento, montagem e implantação destes Componentes CORBA é coletivamente chamado CORBA *Component Model*, ou CCM.

Um dos objetivos de CORBA tem sido a descrição de interfaces, que são contratos no lado do cliente. Com a adoção do POA foram dados importantes passos na definição de como as implementações (serventes) devem ser construídas e gerenciadas pelo ORB. O CCM é o próximo passo, estendendo a IDL com a habilidade de descrever Componentes CORBA e introduz a *Component Implementation Definition Language* (CIDL) para descrever detalhes

de implementação suficientes para permitir que todo o arcabouço do lado do servidor seja gerado, empacotado e implantado.

CCM também define um mapeamento com *Enterprise JavaBeans 1.1* [vide Apêndice B] para permitir que um EJB [Thom98] seja visto como um componente CORBA e vice-versa. Grande parte da especificação do CCM baseia-se em experiências que pessoas têm tido com os dois principais modelos de componentes, MTS (hoje COM+) [COM03] e, principalmente, EJB. Outra fonte de informações foram as experiências no desenvolvimento de ferramentas e arcabouços para construção de servidores CORBA.

CCM é composto por diversas partes que se integram e relacionam:

- Modelo de Componentes Abstrato – extensões à IDL e ao modelo de objetos de CORBA;
- *Component Implementation Framework* (CIF) – *Component Implementation Definition Language* (CIDL);
- Modelo de Programação de Contêineres – descreve o ambiente de desenvolvimento de servidores, apresentando também a visão do cliente;
- Arquitetura de Contêineres – apesar de não fazer parte da especificação do CCM, edições anteriores à versão final desta apresentam uma sugestão para a arquitetura dos contêineres;
- Integração com serviços de transações, persistência e eventos;
- Integração com *Enterprise JavaBeans*.

`component` é um novo meta-tipo básico de CORBA, sendo uma extensão e especialização do meta-tipo `object`. Os componentes são especificados em IDL e podem ser representados no Repositório de Interfaces.

Um componente deve encapsular sua representação interna e implementação, sendo opaco ao cliente e exibindo em sua "superfície" funcionalidades com as quais os clientes, outros serviços de CORBA e elementos do ambiente irão interagir, denominadas portas (*ports*). A Figura 3.1 ilustra a definição em IDL de um componente e a Figura 3.2 apresenta sua representação.

Uma referência a uma instância do componente `Foo` (da Figura 3.1) aparece para os clientes como um objeto CORBA tradicional. Assim, clientes que desconheçam a presença de componentes (*component-unaware*) [seção 5.3, "Considerações sobre a Programação de Clientes"] podem invocar operações através de uma referência à interface equivalente, que identifica unicamente a instância do componente, podendo herdar de outras interfaces, chamadas admitidas (*supported interfaces*).

Vimos que é difícil estender objetos CORBA apenas por herança. Como alternativa, CCM define o conceito denominado facetas (*facets*), semelhante às

interfaces dos componentes COM [BOX98], admitindo interfaces não relacionadas entre si (*unrelated interfaces*), através de composição. Os clientes podem navegar pelas interfaces de um componente (facetas e interfaces equivalentes) por intermédio da interface `Navigation`, definida em `CORBA::CCMObject` (herdada por todos os componentes), ou seja, a partir de uma referência a alguma interface disponibilizada pelo componente é possível obter referência a qualquer de suas interfaces.

```
                                        IDL
interface A, B;
component Foo supports A, B  // definição de interface equivalente (Foo)
{                            // e de interfaces (A e B) admitidas por
                             // herança (supported interfaces)
      provides W, X, Y, Z;   // Facetas (provided interfaces)
      . . .                  // outras definições do componente
}
```

Figura 3.1: representação em IDL de um componente
com seis interfaces, sendo duas disponíveis através de herança (A e B) e
as demais por composição (W, X, Y e Z).

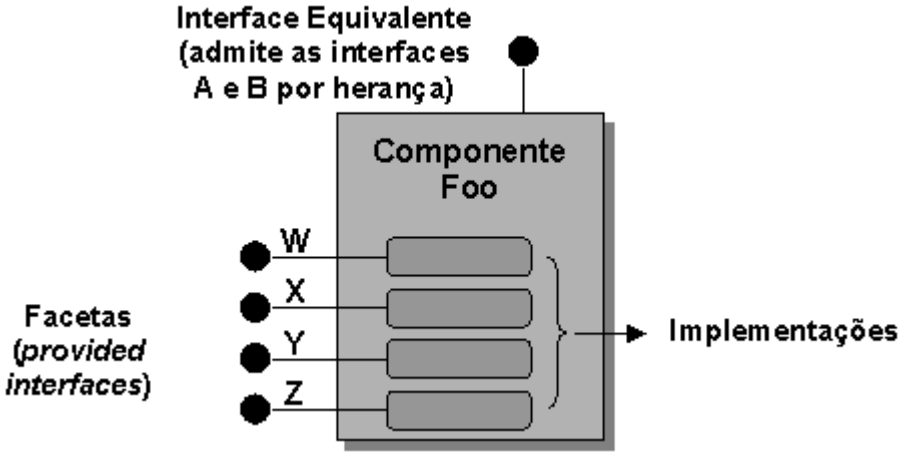

Figura 3.2: representação do componente da Figura 3.1.
Ilustra a opacidade da implementação das interfaces.

As facetas representam um dos quatro tipos de portas, sendo fornecidas para interação com os clientes. Os outros tipos são:

- Receptáculos (*Receptacles*): pontos de conexão que permitem que um componente utilize referências fornecidas por agentes externos;
- Fontes de eventos (*Event sources*): pontos de conexão que emitem eventos (publicação) para um ou mais consumidores de eventos, ou para um canal de eventos;
- Consumidores de eventos (*Event sinks*[2]): pontos de conexão para subscrição a eventos.

[2] Aqui optamos pela tradução consumidores de eventos, uma vez que a definição destes em IDL é realizada por meio da palavra-chave `consumes` [seção 4.7.6, "Consumidores de Eventos (*Event Sinks*)"].

A Figura 3.3 a seguir ilustra os quatro tipos de portas do CCM.

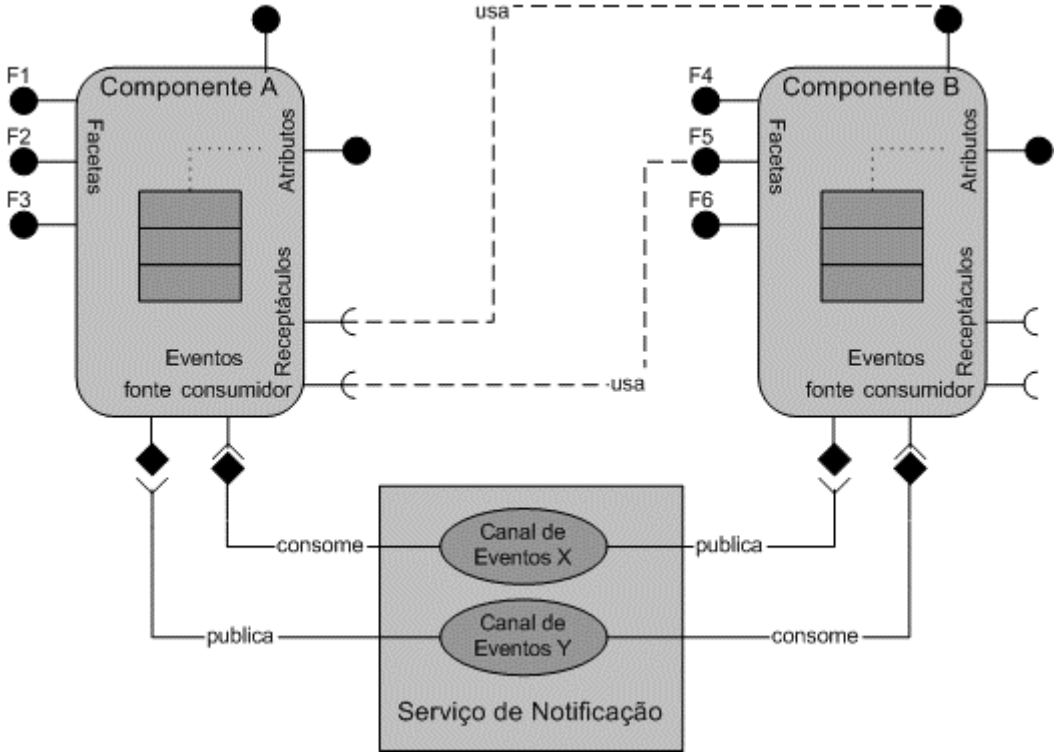

Figura 3.3: os tipos de portas do CCM

Existem, ainda, outros elementos que compõem o modelo de componentes, que são:

- Atributos (*Attributes*): facilitam a configuração dos componentes, permitindo operações de acesso e atribuição;
- Chaves primárias (*Primary keys*): valores expostos a clientes para identificar uma determinada instância de um componente. Apenas os componentes que precisam ser localizados por operações *Finder*, chamados *keyed components*, possuem chaves primárias;
- Interface `Home`, que fornece operações padronizadas para *Factory* e *Finder* [GHJV94]. O *home* de um componente age como um gerenciador para instâncias deste. Algumas tarefas do *home* são o gerenciamento do ciclo de vida do componente e a associação entre chaves primárias e instâncias.

3.1 Exemplo de Utilização

Uma das principais contribuições do CCM é a padronização do processo de desenvolvimento, que pode ser resumido como:

- O desenvolvedor de componentes define, em IDL, as interfaces que as implementações dos componentes irão oferecer;
- O desenvolvedor implementa os componentes utilizando ferramentas disponibilizadas pelo fornecedor do CCM;
- O componente é empacotado em uma DLL;
- O componente é implantado por mecanismo desenvolvido pelo fornecedor do CCM, em um servidor de componentes (*component server*), que é um processo responsável por carregar as DLLs associadas às implementações dos componentes.

Neste trabalho utilizaremos um exemplo que ilustra a definição e implementação de componentes, bem como a utilização de vários conceitos do CCM, como facetas, receptáculos, publicação e subscrição de eventos, composições, empacotamento e a implantação de componentes em contêineres.

O código fonte completo do exemplo pode ser encontrado no Apêndice C.

3.1.1 Descrição do Problema

A aplicação trata do fluxo de notícias, indo desde sua fonte até o leitor.

Uma vez que um jornalista identificar uma notícia, deve publicá-la em uma central de notícias. Esta central, então, envia a notícia a seus assinantes.

Este fluxo deve obedecer às seguintes condições:

- Alguns jornalistas podem trabalhar com exclusividade para uma única central de notícias. Outros, entretanto, podem enviar suas notícias a mais de uma central;
- Para enviar uma notícia, o jornalista deve estar conectado à central;
- Os assinantes podem ou não estar conectados à central de notícias.

3.1.2 A Solução

Para resolver o problema, modelamos quatro componentes CORBA:

- Jornalista: possui uma interface onde o usuário descreve a notícia e a envia a um publicador, ao qual esteja associado;
- JornalistaMult: semelhante ao componente anterior, permitindo entretanto que um jornalista envie notícias a mais de um publicador simultaneamente;

- Publicador: recebe as notícias vindas dos jornalistas e as distribui aos assinantes;
- Assinante: recebe e exibe as notícias enviadas pelo publicador.

A interface entre os jornalistas e os publicadores é feita por meio de portas do tipo faceta e receptáculo, adequadas para comunicação síncrona, onde os componentes devem estar conectados. Assim, se o publicador não estiver disponível no momento do envio de notícia pelo jornalista, ocorrerá uma exceção.

Para a interface entre os publicadores e seus assinantes optamos por portas de eventos, permitindo comunicação assíncrona, ou seja, os assinantes não precisam estar conectados para que o publicador envie a notícia. Dessa forma, se um ou mais assinantes estiverem indisponíveis quando da publicação de uma notícia, não ocorrerá exceção.

A figura a seguir ilustra as relações existentes entre os componentes:

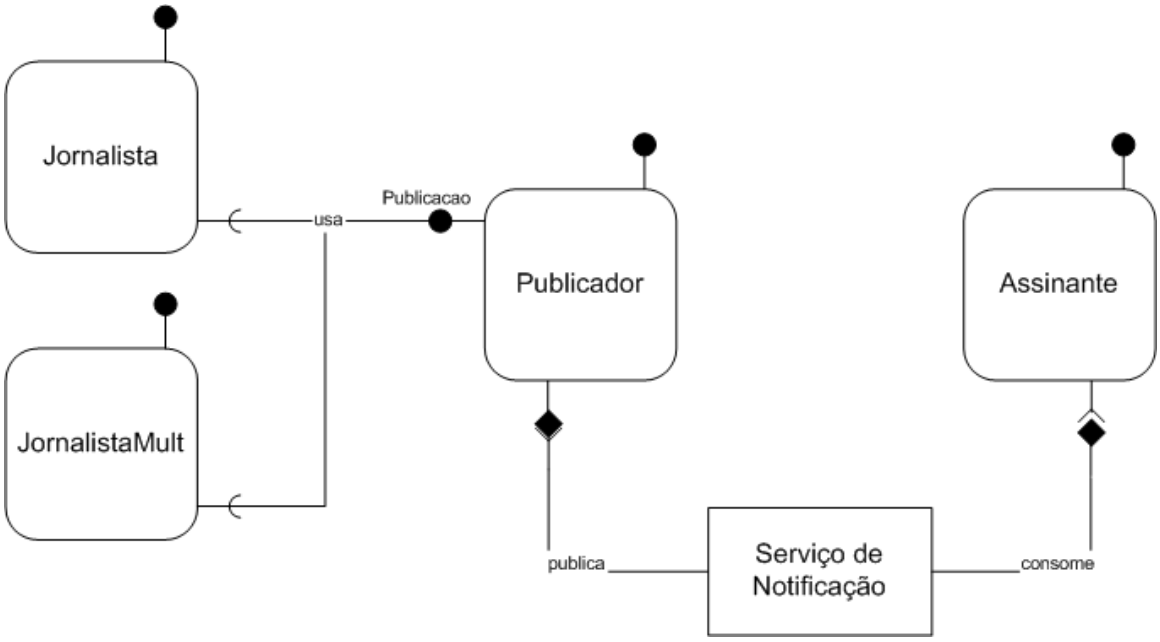

Figura 3.4: componentes CORBA

Neste exemplo utilizaremos dois publicadores. Cada um deles enviará notícias a três assinantes, e receberá notícias de três jornalistas com exclusividade. Existe ainda um outro jornalista, que enviará notícias aos dois publicadores. A figura a seguir ilustra as instâncias dos componentes a serem criadas.

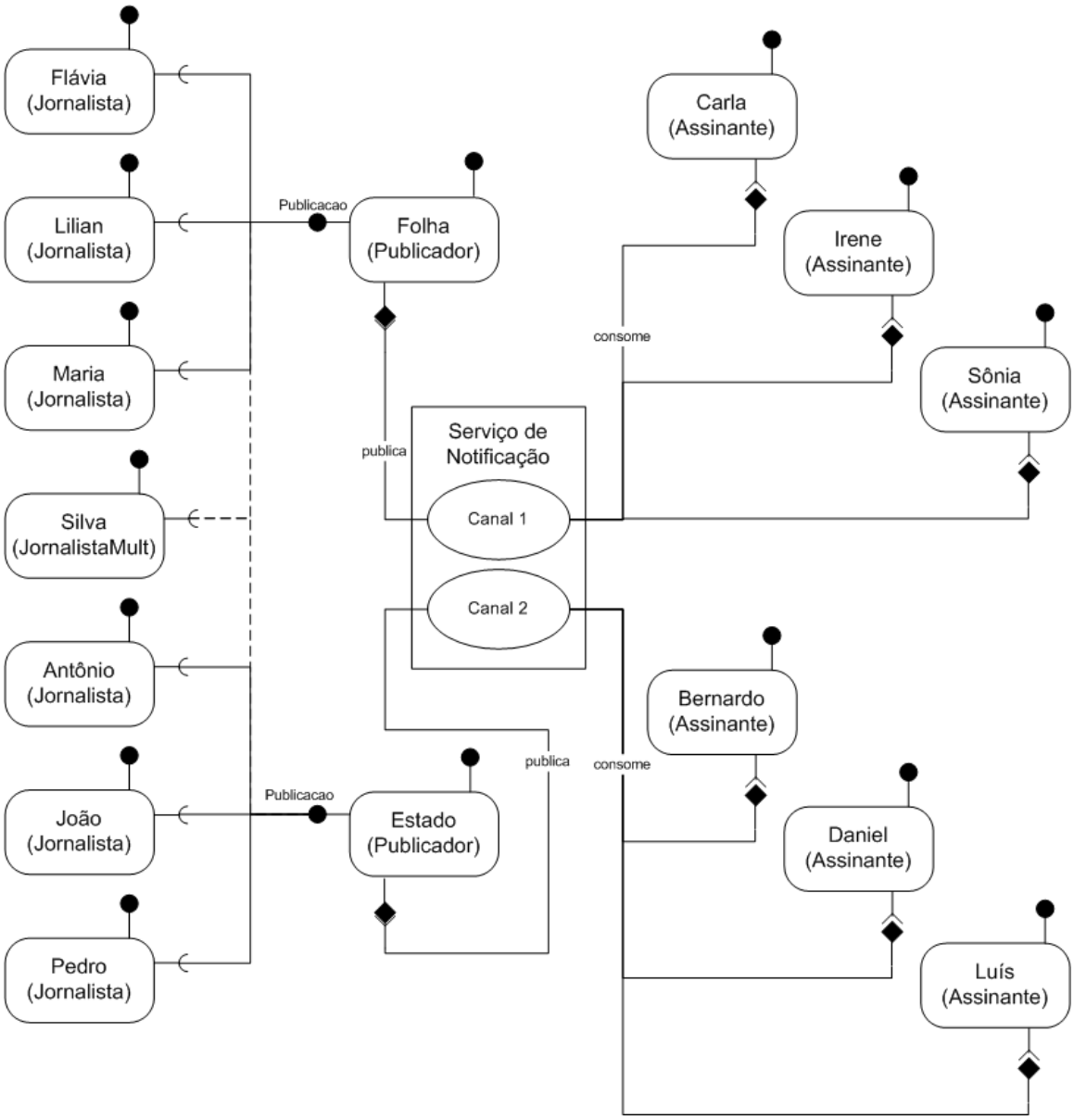

Figura 3.5: exemplo de utilização dos componentes da Figura 3.4
A instância “Silva” envia notícias aos dois publicadores, representado pela linha tracejada.

A Figura 3.6 apresenta o resultado da execução do exemplo.

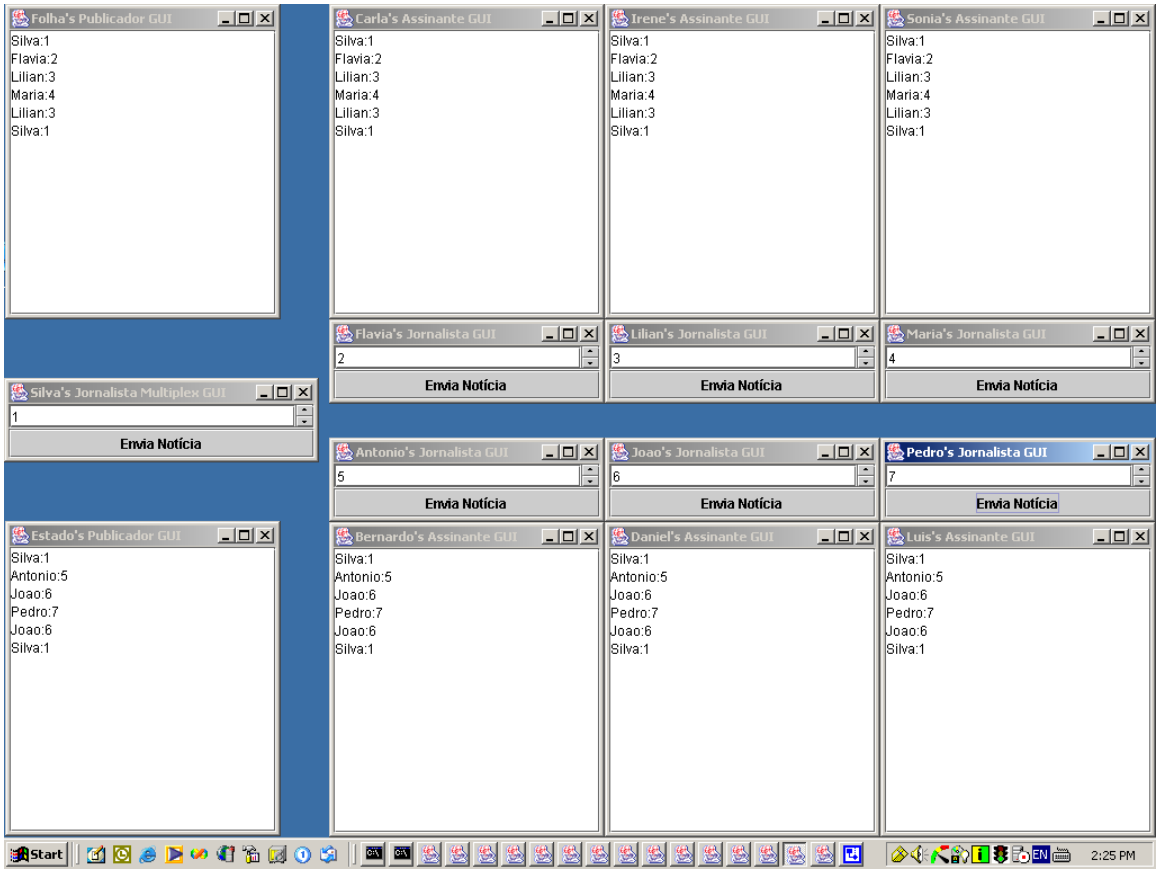

Figura 3.6: execução do exemplo

3.1.3 Ferramentas Utilizadas

Durante este estudo, foram analisadas duas implementações do CCM, em suas primeiras versões, ainda beta. Por serem implementações incipientes, tratam-se de soluções incompletas e com um grau significativo de erros.

A primeira implementação empregada foi o OpenCCM [GOAL01], em conjunto com o ORBacus [IONA03]. Problemas como o uso de receptáculos *multiplex* e chaves primárias consumiram tempo e esforço consideráveis. A decisão por mudar de implementação, contudo, veio do fato desta implementação não fornecer um compilador para CIDL, o que limitou as possibilidades de testes a realizar.

Neste momento foi lançada a implementação EJCCM [EJCCM03], com mais recursos e facilidades quando comparada ao OpenCCM. Decidimos então pela utilização desta implementação, na versão 0.2.2, em conjunto com o ORB OpenORB [COOP03] versão 1.4.0 (versão CVS de 31/jan/2003), a linguagem Java [JAVA02] e o sistema operacional Windows2000, com Service Pack 2.

Capítulo 4

Modelo de Componentes

4.1 Tipos de Componentes

Um componente CORBA pode ser classificado como básico ou estendido. Ambos são gerenciados por c*omponent home*, mas diferem nas funcionalidades que disponibilizam.

Um componente básico permite que objetos CORBA já existentes sejam componentizados. Além disso, há grande similaridade, em termos de funcionalidades, com EJBs, o que contribui significativamente para soluções em que haja integração entre ambos. Esse tipo de componente, entretanto, não é capaz de disponibilizar facetas, receptáculos, consumidores ou fontes de eventos. Ele apenas disponibiliza atributos e a interface equivalente.

No caso de um componente estendido, um conjunto mais amplo de funcionalidades é oferecido, como qualquer tipo de porta, por exemplo.

Outras características de cada tipo de componente serão citadas oportunamente no texto.

4.2 Definição de Componentes

A definição de um componente é expressa por meio de uma declaração em IDL, que se divide em duas partes: um cabeçalho, denotado pela palavra-chave `component`, e um corpo, que é o conteúdo delimitado por "`{`" e "`}`".

A compilação dessa definição do componente gerará uma interface contendo as funcionalidades presentes no corpo da definição do componente.

Tal interface estende o conceito da definição de interfaces CORBA de modo a dar suporte às novas funcionalidades oferecidas pelo CCM, tais como portas e atributos.

A existência, na IDL do CORBA3, da construção `component`, ao invés de apenas incluir novos elementos na construção `interface`, torna a gramática elegante e clara, isolando elementos referentes apenas a componentes. Além disso, existem características que diferem para componentes e interfaces como, por exemplo, o fato dos componentes darem suporte a herança simples, enquanto as interfaces admitem herança múltipla.

4.3 Declaração de Componentes

A especificação do CCM não define palavra-chave para diferenciar a declaração de componentes básicos ou estendidos. Essa decisão deveu-se ao receio de que pudesse haver problemas na evolução das especificações do CCM e do EJB [CCM02, nota ao final da seção 1.3.1, "*Basic Components*"]. Apesar disso, todo componente básico possui a seguinte declaração, em IDL[3]:

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

A declaração dos componentes estendidos contém elementos adicionais, como facetas ou uso de herança, por exemplo.

4.3.1 IDL Equivalente

A IDL, com a especificação do CCM, ganhou novas construções para a declaração de componentes. Para cada uma dessas construções existe uma construção equivalente na IDL definida em CORBA 2.x. A esta última denomina-se IDL equivalente. Essa relação deve-se ao fato do meta-tipo `component` estender o meta-tipo `interface`. Pode-se considerar, para fins de compreensão, que o código em IDL3[4] será convertido para a IDL equivalente (em IDL2) e, então, processado pelo compilador de IDL2. De fato, isso é o que faz a implementação utilizada neste trabalho [EJCCM03].

A construção `component`, em particular, corresponde, em IDL2, à construção `interface`. Assim, pode-se dizer que cada componente possui uma interface equivalente. Os exemplos a seguir ilustram diversas declarações de componentes.

[3] Este texto apresenta trechos da gramática inerente ao CCM, extraídos da especificação. O objetivo é ilustrar conceitos específicos sem, entretanto, reproduzir a totalidade da gramática.
[4] Onde necessário, o termo IDL3 será utilizado para representar a IDL como definida na especificação CORBA 3 e IDL2, do mesmo modo, para a especificação CORBA 2.x.

- Declaração simples:

IDL3:

```
component component_name { … };
```

IDL2 (interface equivalente):

```
interface component_name: Components::CCMObject { … };
```

- Interfaces admitidas por herança:

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> { … };
```

4.3.2 Corpo dos Componentes

A declaração de um componente constitui um escopo de nomes (*naming scope*). As declarações dos diversos tipos de portas, presentes no corpo do componente, são representadas por construções na interface equivalente, como detalhado na próxima seção.

4.4 Facetas (*Facets* ou *Provided Interfaces*)

As facetas são mecanismos através dos quais o componente expõe suas funcionalidades aos clientes. Cada componente pode ter zero ou mais facetas. Caso não haja facetas, o componente expõe apenas a interface equivalente aos clientes. O restante desta seção detalha a sintaxe, semântica e funcionalidades das facetas, bem como a navegação entre elas.

4.4.1 Sintaxe

A declaração de facetas é feita no corpo da declaração do componente, por meio da palavra-chave `provides`. A declaração a seguir, em IDL3:

```
provides <interface_type> <name>;
```

resulta na seguinte construção, na IDL equivalente (IDL2):

```
<interface_type> provide_<name> ();
```

4.4.2 Semântica

Clientes que utilizem instâncias de componentes podem obter referências a facetas através da operação `provide_<name>` da IDL equivalente, correspondente à construção `provides` na definição do componente em IDL3.

A implementação do componente é responsável por garantir que:

- A operação `provide_<name>` devolva uma referência à faceta em questão. Entretanto, a critério do implementador do componente, esta operação pode devolver uma referência nula, ou `nil` (em Java, `null`);
- A referência devolvida pela operação `provide_<name>` dê suporte às operações da interface presente na construção `provides` da declaração do componente. Em particular, quando a operação `_is_a` for invocada nesta referência, recebendo o identificador da faceta no repositório, o resultado deve ser `TRUE`. Tal comportamento pode ser ilustrado pelo código:

```
// comp é uma referência a uma instância do componente
Publicacao faceta_publicacao = comp.provide_faceta_publicacao();
// a operação abaixo devolve TRUE
faceta_publicacao._is_a("IDL:Noticia/Publicacao:1.0")
```

4.4.3 Navegação entre Facetas

Para tornar possível a obtenção de referência a uma faceta de um componente, a partir de outra referência, processo chamado navegação, CCM define o seguinte conjunto de operações, fornecidas pelo componente:

- `CORBA::Object::get_component`: devolve referência à interface equivalente do componente a partir de uma faceta (referência);
- `provide_<name>`: devolve referência a uma faceta a partir de referência à interface equivalente;

Há ainda, a interface `Navigation`, estendida pela interface `CCMObject` que, por sua vez, é herdada por todos os componentes. Segue o conjunto de operações da interface `Navigation`:

- `provide_facet`: recebe o nome de uma faceta, e devolve uma referência a esta;
- `get_all_facets`: devolve uma seqüência com objetos contendo, cada um, o `RepositoryID`, o nome e uma referência a cada faceta do componente, inclusive as herdadas. Para os componentes sem facetas, a seqüência devolvida é de comprimento zero;
- `get_named_facets`: devolve seqüência como a da operação `get_all_facets`, porém apenas para as facetas de nome na lista recebida como parâmetro;
- `same_component`: permite determinar se duas referências pertencem ao mesmo componente.

4.5 Interfaces Admitidas por Herança (*Supported Interfaces*)

Uma vez que a interface equivalente para interfaces admitidas é expressa por meio de herança, a obtenção de referência a estas, pelo cliente, deve ser realizada por meio de operações de alargamento (*widening*).

4.6 Receptáculos

A definição de um componente pode descrever sua habilidade de utilizar, de modo síncrono, referências a outros objetos. Assim, a cada chamada de operação do objeto referenciado, o qual deve ter sido previamente instanciado, o controle da execução volta ao componente apenas após a conclusão da operação chamada. Eventos, por outro lado, possuem comportamento assíncrono, sendo que o consumidor não precisa estar instanciado no momento da chamada e o consumo do evento pode ser realizado *a posteriori*, sem bloquear o componente. Especificamente, essas referências são facetas de

outros componentes. Assim, em tempo de execução, dizemos que o receptáculo de um componente está conectado a uma (*simplex*) ou mais (*multiplex*) faceta(s) de outro componente.

O conceito de receptáculo contribui na apresentação de um nível de abstração mais alto, onde a declaração de um componente explicita o que este oferece (facetas) e utiliza (receptáculos).

Um componente pode possuir zero ou mais receptáculos.

4.6.1 Sintaxe

A declaração de um receptáculo é feita no corpo da declaração do componente, por meio da palavra-chave `uses`. A declaração a seguir, em IDL3, especifica um receptáculo que se conecta a uma única faceta (*simplex*):

```
uses <interface_type> <receptacle_name>;
```

resulta no seguinte conjunto de operações, na IDL equivalente (IDL2):

```
void connect_<receptacle_name> ( in <interface_type> conxn ) raises (
   Components::AlreadyConnected,
   Components::InvalidConnection );
<interface_type> disconnect_<receptacle_name> ( )
   raises ( Components::NoConnection );
<interface_type> get_connection_<receptacle_name> ( );
```

A declaração de receptáculos que se conectam a múltiplas facetas (*multiplex*) possui a seguinte forma:

```
uses multiple <interface_type> <receptacle_name>;
```

que resulta no seguinte conjunto de operações, na IDL equivalente (IDL2):

```
struct <receptacle_name>Connection {
   <interface_type> objref;
   Components::Cookie ck;
};
sequence <receptacle_name>Connection <receptacle_name>Connections;
Components::Cookie connect_<receptacle_name> ( in <interface_type> connection )
raises (
   Components::ExceededConnectionLimit,
   Components::InvalidConnection
);
<interface_type> disconnect_<receptacle_name> (in Components::Cookie ck)
   raises ( Components::InvalidConnection );
<receptacle_name>Connections get_connections_<receptacle_name> ( );
```

4.6.2 Semântica

Em linhas gerais, as operações nos receptáculos independem destes serem *simplex* ou *multiplex*. Entretanto, existem comportamentos específicos para cada um dos tipos. Sendo assim, a lista a seguir foi estruturada de modo a apresentar, para cada operação, os comportamentos de cada tipo de receptáculo.

- `connect_<receptacle_name>`: essas operações devem ser implementadas em parte pelo programador e em parte pelo CIF (vide Capítulo 6). O receptáculo mantém uma cópia da referência a objeto recebida como parâmetro, através da qual pode invocar operações.
    - *Simplex*: pode haver apenas uma conexão, sendo que novas tentativas de conectar incorrerão na exceção `AlreadyConnected`. Outra funcionalidade é a habilidade do componente recusar, por motivos arbitrários, pedidos de conexão. Ao fazer isso, deve disparar a exceção `InvalidConnection`;
    - *Multiplex*: têm a capacidade de manter várias conexões. Se desejado, a implementação pode limitar o número de conexões, sendo que o componente deve disparar a exceção `ExceededConnectionLimit`.
- `disconnect_<receptacle_name>`: essas operações encerram o relacionamento entre o receptáculo e a referência a objeto a ele associada.
    - *Simplex*: se houver conexão, devolve a referência ao objeto . Caso contrário, dispara a exceção `NoConnection`;
    - *Multiplex*: a operação recebe um parâmetro do tipo `Components::Cookie`, que deve ter sido devolvido pela operação `connect_<receptacle_name>`. É responsabilidade do cliente associar *cookies* a referências a objetos. Caso o parâmetro em questão não se refira à referência correta, a operação de desconexão deve disparar a exceção `InvalidConnection`.
- `get_connection_<receptacle_name>`: apenas para receptáculos *simplex*. Quando houver conexão, devolve referência ao objeto conectado. Caso contrário, devolve referência nula;
- `get_connections_<receptacle_name>`: apenas para receptáculos *multiplex*. Devolve uma seqüência onde cada elemento contém uma referência a um objeto conectado e o valor do *cookie* associado. Não havendo conexões, a seqüência terá comprimento zero.

4.6.3 A Interface `Receptacles`

As duas últimas seções apresentaram operações específicas para cada receptáculo declarado. No processo de compilação, tais operações farão parte da interface equivalente do componente. A interface `Receptacles`, por sua vez, fornece operações genéricas para conexão aos receptáculos de um componente. A interface `CCMObject` deriva da interface `Receptacles`. No caso de um componente básico, as operações genéricas da interface `Receptacles` não têm utilidade prática, uma vez que este tipo de componente não possui receptáculos.

As operações dessa interface são:

- `connect`: conecta a referência a objeto recebida como parâmetro ao receptáculo cujo nome também foi recebido como parâmetro. Para receptáculos *simplex*, devolve referência nula. Para os *multiplex*, devolve um *cookie* para ser usado na desconexão;
- `disconnect`: para um receptáculo *simplex*, desconecta-o da referência a objeto a ele associada. No caso de um receptáculo *multiplex*, desconecta-o da referência a objeto associada a ele, correspondente ao *cookie* recebido como parâmetro;
- `get_connections`: devolve uma seqüência de estruturas `ConnectionDescription`, onde cada uma contém uma referência a objeto e o *cookie* correspondente;
- `get_all_receptacles`: devolve uma seqüência de estruturas `ReceptacleDescription`, onde cada uma contém dados sobre um dos receptáculos do componente, inclusive os herdados, e indicação do tipo, *simplex* ou *multiplex*;
- `get_named_receptacles`: devolve seqüência como a da operação `get_all_receptacles`, porém apenas para os receptáculos de nome na lista recebida como parâmetro.

4.7 Eventos

O CCM oferece tratamento de eventos no modelo *push*, com a publicação/subscrição de eventos. Esse modelo foi idealizado de modo a ser compatível com o Serviço de Notificação CORBA [NSS02]. De fato, as interfaces expostas pelo CCM se traduzem em interface de programação cuja semântica corresponde a um subconjunto da semântica do serviço de notificação.

As implementações do contêiner fornecem serviços para tratamento de eventos para componentes e clientes. As implementações dos componentes obtêm referência a tais serviços durante a inicialização e tratam o acesso a estes pelos clientes. A implementação do contêiner é livre para fornecer

qualquer mecanismo de eventos, desde que atenda à semântica requerida. O contêiner ainda é responsável pela configuração deste mecanismo, bem como pela qualidade de serviço por este oferecida e pela política de roteamento a ser empregada quando do envio de eventos.

4.7.1 *Event Types*

Para uso pelo CCM, a IDL define a construção `eventtype` para a declaração de eventos. Esta, por sua vez, é uma forma restrita do tipo `valuetype`.

4.7.1.1 Eventos no CCM e o Serviço de Notificação de CORBA

A implementação do mecanismo de eventos dos Componentes CORBA é feita sobre o Serviço de Notificação CORBA. Assim sendo, um evento, declarado como `eventtype`, deve ser inserido em uma instância do tipo `any`. Do mesmo modo, o resultado deve ser inserido na estrutura de eventos esperada pelo componente.

O mapeamento entre os eventos do CCM e o Serviço de Notificação CORBA é de responsabilidade do contêiner (seção 5.2.2.3, “Eventos e Notificação”).

4.7.1.2 Sintaxe

A declaração de um *event type* é feita independentemente da declaração do componente, por meio da palavra-chave `eventtype`. A declaração a seguir, em IDL3:

```
module <module_name> {
   valuetype A { <A_state_members> };
   eventtype B : A { <B_state_members> };
   eventtype C : B { <C_state_members> };
};
```

resulta no seguinte conjunto de declarações, na IDL equivalente (IDL2):

```
module <module_name> {
   valuetype A { <A_state_members> };
   valuetype B : A, ::Components::EventBase {<B_state_members> };
   interface BConsumer : ::Components::EventConsumerBase {
      void push_B (in B the_b);
   };
   valuetype C : B {  <C_state_members> };
   interface CConsumer : BConsumer {
      void push_C (in C the_c);
   };
```

```
};
```

A declaração do `eventtype B` em IDL3 resultou um `valuetype` em IDL2 com a presença da interface `::Components::EventBase` na cadeia de herança, além da interface consumidora, a partir de `::Components::EventConsumerBase`. Vale notar que as interfaces consumidoras estão na mesma relação de herança dos *event types* onde se originaram.

4.7.1.3 A Interface `EventBase`

O objetivo desta interface é servir de base para a definição de `eventtypes`, mapeados para `valuetypes` em IDL2, e possui a seguinte declaração, em IDL3:

```
module Components {
   abstract valuetype EventBase { };
};
```

4.7.2 A Interface `EventConsumerBase`

O modelo *push* adotado pelo CCM possui mecanismos básicos definidos por interfaces consumidoras. As fontes de eventos mantêm referências a consumidores e invocam várias formas de operações *push* para enviar eventos.

As interfaces para consumo de eventos são derivadas da interface `Components::EventConsumerBase`, que é definida como:

```
module Components {
   exception BadEventType {  CORBA::RepositoryId expected_event_type;  };
   interface EventConsumerBase {
      void push_event(in EventBase evt) raises (BadEventType);
   };
};
```

A operação `push_event` envia o evento `evt` ao consumidor e é implementada pelo CCM. Por ser uma operação genérica, ao contrário das operações ilustradas na seção 4.7.1.2, que são específicas a cada tipo de evento, pode haver necessidade de restringir quais eventos o consumidor pode aceitar. Nesse caso, a implementação gerada pelo CCM pode ser alterada para que a exceção `BadEventType` seja disparada quando ocorrer um evento indesejado pelo consumidor. O membro `expected_event_type` desta exceção contém o `RepositoryID` do tipo esperado pelo consumidor.

Vale notar que essa exceção pode ser disparada apenas pelo consumidor cuja referência foi utilizada para chamar a operação `push_event`. É possível que um consumidor seja utilizado como um *proxy* para a propagação de eventos a subscritores, seja diretamente ou através de canal de notificação. Se algum desses subscritores disparar quaisquer exceções, estas não serão propagadas de volta à fonte original do evento, ou seja, o componente.

4.7.3 Fontes de Eventos (*Event Sources*)

Uma fonte de eventos oferece a um componente a capacidade de gerar eventos de um tipo específico. Fornece, ainda, mecanismos para associá-lo a consumidores, através de canais de eventos fornecidos pelo contêiner.

Existem duas categorias de fontes de eventos: emissores (*emitters*) e publicadores (*publishers*). Um emissor pode ser associado a até um consumidor de eventos. Um publicador, por outro lado, pode ser associado a um número arbitrário de consumidores, a que se denomina subscrição. Neste caso, dizemos que consumidores subscrevem-se a publicadores.

4.7.4 Emissores (*Emitters*)

Um emissor possui as seguintes características:

- As operações equivalentes para os emissores permitem que apenas um consumidor esteja associado a um emissor por vez;
- Os eventos disparados por um emissor são enviados ao canal de eventos fornecido pelo contêiner em tempo de execução;
- Um mesmo canal de eventos pode ser compartilhado por várias fontes de eventos;
- O contêiner envia os eventos através do canal de eventos a um consumidor de eventos específico, determinado pelo parâmetro da operação de conexão (ver seções 4.7.4.1 e 4.7.4.2);
- Embora os emissores possam ser utilizados pelos clientes, seu propósito é o de serem utilizados para a configuração de componentes.

4.7.4.1 Sintaxe

A declaração de um emissor é feita no corpo da declaração do componente, por meio da palavra-chave `emits`. A declaração a seguir, em IDL3:

```
module <module_name> {
```

```
   component <component_name> {
      emits <event_type> <source_name>;
   };
};
```

resulta no seguinte conjunto de declarações, na IDL equivalente (IDL2):

```
module <module_name> {
   interface <component_name> : Components::CCMObject {
      void connect_<source_name> (in <event_type>Consumer consumer)
         raises (Components::AlreadyConnected);
      <event_type>Consumer disconnect_<source_name>()
         raises (Components::NoConnection);
   };
};
```

4.7.4.2 Semântica

As operações de uma fonte de eventos do tipo emissor permitem a conexão ou desconexão de consumidores de eventos:

- `connect_<source_name>`: conecta o consumidor recebido como parâmetro à fonte de eventos. Se este já estiver conectado a outro consumidor, a exceção `AlreadyConnected` será disparada;
- `disconnect_<source_name>`: desassocia a fonte de eventos do consumidor, e devolve uma referência a este último. Se não havia conexão, dispara a exceção `NoConnection`.

4.7.5 Publicadores (*Publishers*)

Uma fonte de eventos do tipo publicador possui as seguintes características:

- As operações equivalentes para os publicadores permitem que múltiplos subscritores (consumidores) estejam conectados simultaneamente;
- Assim como ocorre para os emissores, os eventos disparados por um publicador são enviados pelo canal de eventos, fornecido pelo contêiner em tempo de execução, aos consumidores subscritos;
- Cada canal de eventos pode ser utilizado por apenas um componente para publicação;
- A finalidade dos publicadores, ao contrário dos emissores, é fornecer aos clientes acesso direto a eventos disparados pelo componente.

4.7.5.1 Sintaxe

A declaração de um publicador é feita no corpo da declaração do componente, por meio da palavra-chave `publishes`. A declaração a seguir, em IDL3:

```
module <module_name> {
   component <component_name> {
      publishes <event_type> <source_name>;
   };
};
```

resulta no seguinte conjunto de declarações, na IDL equivalente (IDL2):

```
module <module_name> {
   interface <component_name> : Components::CCMObject {
      Components::Cookie subscribe_<source_name>
         (in <event_type>Consumer consumer)
         raises (Components::ExceededConnectionLimit);
      <event_type>Consumer unsubscribe_<source_name>
         (in Components::Cookie ck)
         raises (Components::InvalidConnection);
   };
};
```

4.7.5.2 Semântica

As operações de uma fonte de eventos do tipo publicador permitem a conexão (subscrição) ou desconexão de consumidores de eventos:

- `subscribe_<source_name>`: conecta o consumidor recebido como parâmetro ao canal de eventos. Por definição, o componente deve ser o único publicador de eventos para o canal. Se a implementação do componente ou o canal estipularem número máximo de conexões simultâneas e este for ultrapassado, a exceção `ExceededConnectionLimit` será disparada. O *cookie* devolvido identifica a subscrição que associa o subscritor ao publicador, e deve ser utilizado quando da invocação da operação `unsubscribe_<source_name>`;
- `unsubscribe_<source_name>`: desassocia a subscrição, identificada pelo *cookie* recebido como parâmetro, entre o subscritor e o publicador. Se o *cookie* não identificar uma conexão, será disparada a exceção `InvalidConnection`.

4.7.6 Consumidores de Eventos (*Event Sinks*)

Consumidores de eventos são o tipo de porta que possibilita a um componente receber eventos. Um consumidor de eventos pode ser visto como um tipo especial de faceta.

Diferentemente do que ocorre com as fontes de eventos, os consumidores de eventos não distinguem entre subscrição e conexão.

A interface consumidora disponibilizada pelo componente pode ser associada a um número arbitrário de fontes de eventos. Além disso, um componente não possui controle sobre quais fontes de eventos têm permissão para enviar eventos às suas interfaces consumidoras. Se houver necessidade deste controle, a interface consumidora não deve ser exposta pelo componente como um consumidor de eventos. Ao invés disso, pode-se criar uma interface consumidora internamente e explicitamente conectá-la ou subscrevê-la às fontes desejadas.

Ao expor consumidores de eventos um componente está, de fato, expressando a capacidade de receber eventos de fontes arbitrárias. Um componente pode exibir zero ou mais consumidores.

4.7.6.1 Sintaxe

A declaração de um consumidor de eventos é feita no corpo da declaração do componente, por meio da palavra-chave `consumes`. A declaração a seguir, em IDL3:

```
module <module_name> {
   component <component_name> {
      consumes <event_type> <sink_name>;
   };
};
```

resulta no seguinte conjunto de declarações, na IDL equivalente (IDL2):

```
module <module_name> {
   interface <component_name> : Components::CCMObject {
      <event_type>Consumer get_consumer_<sink_name>();
   };
};
```

4.7.6.2 Semântica

A operação `get_consumer_<sink_name>` devolve uma referência que dá suporte à interface consumidora específica para o tipo de evento declarado.

4.7.7 A Interface `Events`

As interfaces para tratamento de eventos até aqui citadas fornecem operações específicas para cada fonte ou consumidor de eventos. A interface `Events`, ao contrário, fornece acesso genérico a fontes e consumidores de eventos em um componente. A interface `CCMObject` deriva da interface `Events`. Para componentes básicos, que não possuem a capacidade de utilizar eventos, isto é, de expor fontes ou consumidores de eventos, apenas as operações genéricas da interface `Events` estão disponíveis.

4.7.7.1 Sintaxe

A declaração da interface `Events`, em IDL3, é:

```
module Components {
   exception InvalidName { };
   exception InvalidConnection { };
   exception AlreadyConnected { };
   exception NoConnection { };
   valuetype ConsumerDescription : PortDescription {
      public EventConsumerBase consumer;
   };
   typedef sequence<ConsumerDescription> ConsumerDescriptions;
   valuetype EmitterDescription : PortDescription {
      public EventConsumerBase consumer;
   };
   typedef sequence<EmitterDescription> EmitterDescriptions;
   valuetype SubscriberDescription {
      public Cookie ck;
      public EventConsumerBase consumer;
   };
   typedef sequence<SubscriberDescription> SubscriberDescriptions;
   valuetype PublisherDescription : PortDescription {
      public SubscriberDescriptions consumers;
   };
   typedef sequence<PublisherDescription> PublisherDescriptions;
   interface Events {
      EventConsumerBase get_consumer (in FeatureName sink_name)
      raises (InvalidName);
      Cookie subscribe (in FeatureName publisher_name,
         in EventConsumerBase subscriber)
         raises (InvalidName,InvalidConnection,ExceededConnectionLimit);
      void unsubscribe (in FeatureName publisher_name,in Cookie ck)
         raises (InvalidName, InvalidConnection);
      void connect_consumer
         (in FeatureName emitter_name,in EventConsumerBase consumer)
         raises (InvalidName, AlreadyConnected,InvalidConnection);
      EventConsumerBase disconnect_consumer (in FeatureName source_name)
         raises (InvalidName, NoConnection);
      ConsumerDescriptions get_all_consumers ();
      ConsumerDescriptions get_named_consumers (in NameList names)
         raises (InvalidName);
      EmitterDescriptions get_all_emitters ();
      EmitterDescriptions get_named_emitters (in NameList names)
```

```
        raises (InvalidName);
      PublisherDescriptions get_all_publishers ();
      PublisherDescriptions get_named_publishers (in NameList names)
        raises (InvalidName);
   };
};
```

4.7.7.2 Semântica

As operações da interface `Events` são:

- `get_consumer`: devolve uma interface `EventConsumerBase` para o consumidor identificado pelo parâmetro `sink_name`. Se este for inválido, a exceção `InvalidName` será disparada;
- `subscribe`: associa o subscritor definido pelo parâmetro `subscriber` à fonte de eventos identificada pelo parâmetro `publisher_name`. Caso este último não seja válido, a exceção `InvalidName` será disparada. Se a referência em `subscriber` não admitir a fonte de eventos, a exceção `InvalidConnection` será disparada. Ainda, se o limite de número de conexões, estabelecido na implementação, for ultrapassado, a exceção `ExceededConnectionLimit` será disparada;
- `unsubscribe`: desassocia o subscritor associado ao parâmetro `ck` da fonte de eventos identificada pelo parâmetro `publisher_name`. Caso este não seja válido, a exceção `InvalidName` será disparada. Se o parâmetro `ck` não identificar um subscritor, a exceção `InvalidConnection` será disparada;
- `connect_consumer`: associa o consumidor denotado pelo parâmetro `consumer` à fonte de eventos identificada pelo parâmetro `emitter_name`. Se este não for válido, a exceção `InvalidName` será disparada. Caso já exista conexão ao emissor, a exceção `AlreadyConnected` será disparada. Se a referência em `consumer` não admitir a fonte de eventos, a exceção `InvalidConnection` será disparada;
- `disconnect_consumer`: desassocia o consumidor correntemente conectado ao emissor denotado pelo parâmetro `emitter_name`. Se este não for válido, a exceção `InvalidName` será disparada. Se nenhum consumidor estiver associado ao emissor, a exceção `NoConnection` será disparada;
- `get_all_consumers`: devolve informações sobre todas as portas consumidoras na cadeia de herança do componente, como uma seqüência de valores do tipo `ConsumerDescription`. A ordem em que tais valores aparecem na seqüência não é especificado. Para componentes que não consomem eventos, tais como os

componentes básicos, a seqüência devolvida possui comprimento zero;

- `get_named_consumers`: devolve informações sobre as portas consumidoras com nome presente na lista identificada pelo parâmetro `names`, como uma seqüência de valores do tipo `ConsumerDescription`. A ordem em que tais valores aparecem na seqüência não é especificada. Se qualquer nome em `names` não for válido na cadeia de herança do componente, a exceção `InvalidName` será disparada;
- `get_all_emitters`: devolve informações sobre todas as portas emissoras na cadeia de herança do componente, como uma seqüência de valores do tipo `EmitterDescription`. A ordem em que tais valores aparecem na seqüência não é especificada. Para componentes que não emitem eventos, tais como os componentes básicos, a seqüência devolvida possui comprimento zero;
- `get_named_emitters`: devolve informações sobre as portas emissoras com nome presente na lista identificada pelo parâmetro `names`, como uma seqüência de valores do tipo `EmitterDescription`. A ordem em que tais valores aparecem na seqüência não é especificada. Se qualquer nome em `names` não for válido na cadeia de herança do componente, a exceção `InvalidName` será disparada;
- `get_all_publishers`: devolve informações sobre todas as portas publicadoras na cadeia de herança do componente, como uma seqüência de valores do tipo `PublisherDescription`. A ordem em que tais valores aparecem na seqüência não é especificada. Para componentes que não publicam eventos, tais como os componentes básicos, a seqüência devolvida possui comprimento zero;
- `get_named_publishers`: devolve informações sobre as portas publicadoras com nome presente na lista identificada pelo parâmetro `names`, como uma seqüência de valores do tipo `PublisherDescription`. A ordem em que tais valores aparecem na seqüência não é especificada. Se qualquer nome em `names` não for válido na cadeia de herança do componente, a exceção `InvalidName` será disparada.

4.8 Herança de Componentes

A cada declaração de componente em IDL3 corresponde uma interface equivalente em IDL2, conforme descrito na seção 4.3.1, “IDL Equivalente”. Assim, o funcionamento da herança de componentes é definido de acordo com as relações de herança entre as interfaces equivalentes dos componentes.

As seguintes regras se aplicam à herança de componentes:

- As interfaces para tipos de componentes não derivados, derivam de `CCMObject`:

IDL3:

```
component <component_name> { … };
```

IDL2 (interface equivalente):

```
interface <component_name>: Components::CCMObject { … };
```

- Se um tipo de componente admite interfaces IDL por herança, a interface equivalente é derivada de `CCMObject` e das interfaces admitidas:

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> { … };
```

- Um tipo de componente derivado não pode admitir interfaces por herança diretamente. Admite apenas as interfaces admitidas pelo componente base. Assim, a construção a seguir é **inválida**:

IDL3:

```
component <component_name> : <base_name>
supports <interface_name_1>, <interface_name_2> { … };
```

- A interface equivalente de um tipo de componente derivado é derivada da interface equivalente do tipo do componente base:

IDL3:

```
component  <base_name> { ... };
component <component_name> : <base_name> { … };
```

IDL2 (interface equivalente):

```
interface <base_name> { ... };
interface <component_name>: <base_name> { … };
```

- Não é possível empregar herança múltipla de componentes, apenas herança simples. Assim, a construção abaixo é **inválida**:

IDL3:

```
component <component_name> : <base_name_1>, <base_name_2> { … };
```

- As portas e atributos de um tipo de componente base são herdados pelo tipo de componente derivado.

4.8.1 A Interface `CCMObject`

A interface `CCMObject`, da qual todos os componentes derivam, é definida por:

```
module Components {
   valuetype ComponentPortDescription {
      public FacetDescriptions facets;
      public ReceptacleDescriptions receptacles;
      public ConsumerDescriptions consumers;
      public EmitterDescriptions emitters;
      public PublisherDescriptions publishers;
   };

   exception NoKeyAvailable { };

   interface CCMObject : Navigation, Receptacles, Events {
      CORBA::IRObject get_component_def ( );
      CCMHome get_ccm_home( );
      PrimaryKeyBase get_primary_key( ) raises (NoKeyAvailable);
      void configuration_complete( ) raises (InvalidConfiguration);
      void remove() raises (RemoveFailure);
      ComponentPortDescription get_all_ports ();
   };
};
```

A semântica das operações de `CCMObject` é:

- `get_component_def`: devolve uma referência à definição do componente no repositório de interfaces. Em linguagens fortemente tipadas, a referência devolvida deve ser estreitada para `CORBA::ComponentIR::ComponentDef` antes que possa ser utilizada;
- `get_ccm_home`: devolve uma referência ao *home* que gerencia o componente;
- `get_primary_key`: devolve uma referência à chave primária associada à instância do componente por seu *home*. Caso não haja chave primária, a exceção `NoKeyAvailable` será disparada;
- `configuration_complete`: esta operação é invocada pelo configurador para indicar o término da fase de configuração (seção 4.11, “Configuração de Componentes”). Caso o componente não esteja pronto para o início da fase operacional, a exceção `InvalidConfiguration` será disparada;
- `remove`: destrói a instância do componente;

- `get_all_ports`: devolve um valor com a descrição de todas as portas do componente.

4.9 *Homes*

O objetivo de um *home* é gerenciar instâncias de um tipo específico de componente. Se um componente demandar o uso de chave primária, é responsabilidade do *home* estabelecer e manter as associações entre os valores de chave primária e as instâncias do componente. As principais características de uma definição de *home*, realizada em IDL, são:

- Semelhantemente ao que ocorre com componentes, a compilação da definição de um *home* produz uma interface equivalente;
- Do mesmo modo, a definição de *home* que contenha chave primária resulta em um conjunto de operações gerado automaticamente na interface equivalente. Estas operações são detalhadas na seção 4.9.4, “Definições de *Homes* Com Chave Primária”.

4.9.1 Chaves Primárias

Chaves primárias identificam unicamente instâncias de componentes gerenciados no escopo de determinado *home*. Assim, não pode haver duas instâncias de componentes gerenciadas pelo mesmo *home* com mesmo valor de chave primária.

Um determinado tipo de chave primária é associado ao *home*, e não a um tipo de componente. Desta forma, é possível criar duas instâncias de um mesmo componente associadas a diferentes tipos de chave primária, desde que sejam gerenciadas por *homes* diferentes. Por exemplo, dado um componente para representar uma pessoa, é possível associar instâncias a tipos de chaves primárias diferentes, como CPF ou nome (considerando que não haja homônimos).

4.9.1.1 Restrições de Chaves Primárias

Um determinado tipo de chave primária:

- Deve ser derivado de `Components::PrimaryKeyBase`;
- Deve ser um tipo concreto com pelo menos um membro público;
- Não pode conter membros privados;
- Não pode conter membros que sejam referências a uma interface CORBA.

Essas restrições aplicam-se recursivamente aos tipos dos membros, que podem ser `struct`, `array`, `sequence` e `union`.

4.9.1.2 O Tipo PrimaryKeyBase

O tipo base para qualquer chave primária é `Components::PrimaryKeyBase`, definido como:

```
module Components {
   abstract valuetype PrimaryKeyBase { };
};
```

4.9.2 Interfaces Explícita, Implícita e Equivalente

A definição de *home* implicitamente produz um conjunto de operações cujos nomes são comuns a todos os *homes*, sendo que as assinaturas diferem em função do tipo de componente gerenciado. Por exemplo, a operação `create`, descrita na seção 4.9.3, está presente para todos os *homes*. Entretanto, o tipo de componente devolvido é aquele gerenciado para cada *home*.

A finalidade das operações possuírem o mesmo nome para diferentes *homes* é fornecer ao cliente do componente um conjunto uniforme de operações para tarefas básicas, relacionadas com o gerenciamento de componentes, tais como criação, busca e destruição. As assinaturas diferem para manter as operações específicas para cada tipo de componente. Devido a essa característica, tais operações não podem ser herdadas, e há um aumento na complexidade do modelo de componentes. Entretanto, o modelo disponível para o programador da aplicação cliente é relativamente simples.

Cada definição de *home* produz três interfaces:

- Explícita, cujo nome é da forma `<nome do home>Explicit`, e que contém as operações definidas explicitamente na definição do *home* (inclusive operações explicitamente definidas para *factory* e *finder*). Contém ainda o conjunto de operações padrão constantes da interface `CCMHome` (seção 4.9.9);
- Implícita, cujo nome é da forma `<nome do home>Implicit`, contendo as operações acima descritas;
- Equivalente, cujo nome é simplesmente `<nome do home>`, que herda das duas anteriores, sendo disponível para o programador que utiliza o *home*.

4.9.3 Definições de *Homes* Sem Chave Primária

A declaração a seguir, em IDL3:

```
home <home_name> manages <component_type> {
   <explicit_operations>
};
```

resulta nas seguintes declarações de interfaces explícita, implícita e equivalente, em IDL2:

```
interface <home_name>Explicit : Components::CCMHome {
   <equivalent_explicit_operations>
};
interface <home_name>Implicit : Components::KeylessCCMHome {
   <component_type> create() raises(CreateFailure);
};
interface <home_name> : <home_name>Explicit, <home_name>Implicit { };
```

Nas declarações acima, `<equivalent_explicit_operations>` são as declarações constantes na definição do *home*, em `<explicit_operations>`, acrescidas das operações de *factory* e *finder* em sua forma equivalente. Essas operações estão na seção 4.9.6, “Considerações sobre Operações Explícitas”.

A operação `create` cria uma instância do componente gerenciado pelo *home*.

4.9.4 Definições de *Homes* Com Chave Primária

A declaração a seguir, em IDL3:

```
home <home_name> manages <component_type> primarykey <key_type> {
   <explicit_operations>
};
```

resulta nas seguintes declarações de interfaces explícita, implícita e equivalente, em IDL2:

```
interface <home_name>Explicit : Components::CCMHome {
   <equivalent_explicit_operations>
};
interface <home_name>Implicit {
   <component_type> create (in <key_type> key)
      raises (Components::CreateFailure, Components::DuplicateKeyValue,
      Components::InvalidKey);
   <component_type> find_by_primary_key (in <key_type> key)
      raises (Components::FinderFailure, Components::UnknownKeyValue,
      Components::InvalidKey);
   void remove (in <key_type> key)
```

```
      raises (Components::RemoveFailure, Components::UnknownKeyValue,
      Components::InvalidKey);
   <key_type> get_primary_key (in <component_type> comp);
};
interface <home_name> : <home_name>Explicit , <home_name>Implicit { };
```

A semântica das operações da interface implícita é:

- `create`: cria uma instância do componente com o valor de chave primária especificado. Devolve uma referência ao componente criado. Se o valor de chave especificado já tiver sido associado a outra instância de componente gerenciado pelo mesmo *home*, a exceção `DuplicateKeyValue` será disparada. Se a chave não for válida, a exceção `InvalidKey` será disparada. Caso ocorra qualquer outra situação irregular, a exceção `CreateFailure` será disparada;
- `find_by_primary_key`: devolve uma referência ao componente identificado pela chave primária recebida como parâmetro. Caso a chave seja válida, mas não identifique componente gerenciado pelo *home*, a exceção `UnknownKeyValue` será disparada. Se a chave não for válida, a exceção `InvalidKey` será disparada. Outras condições de erro dispararão a exceção `FinderFailure`;
- `remove`: destrói o componente associado à chave primária recebida como parâmetro. Chamadas subseqüentes a operações do componente dispararão a exceção de sistema `OBJECT_NOT_EXIST`. Caso a chave seja válida, mas não identifique componente gerenciado pelo *home*, será disparada a exceção `UnknownKeyValue`. Se a chave não for válida, a exceção `InvalidKey` será disparada. Outras condições de erro dispararão a exceção `RemoveFailure`;
- `get_primary_key`[5]: devolve a chave primária associada ao componente recebido como parâmetro.

4.9.5 Interfaces Admitidas por Herança (*Supported Interfaces*)

Assim como ocorre para componentes, uma definição de *home* pode, opcionalmente, admitir, através de herança, uma ou mais interfaces. Neste caso, a implementação do *home* deve conter as implementações das operações dessas interfaces.

Esta capacidade se aplica a *homes* com ou sem chave primária. A sintaxe a seguir, em IDL3, ilustra o segundo caso:

[5] A especificação de Componentes CORBA referenciada neste trabalho não contém documentação a respeito da função `get_primary_key`.

```
home <home_name> supports <interface_name> manages <component_type> {
   <explicit_operations>
};
```

A interface explícita resultante, em IDL2, herda tanto de `CCMHome` quanto das interfaces admitidas:

```
interface <home_name>Explicit : Components::CCMHome, <interface_name> {
   <equivalent_explicit_operations>
};
```

Os clientes são capazes de estreitar (*narrow*) referência a `CCMHome` para qualquer das interfaces admitidas ou expandir (*widen*) referência à interface equivalente ou explícita para uma das interfaces admitidas.

4.9.6 Considerações sobre Operações Explícitas

O corpo da declaração de um *home* pode conter operações, denominadas explícitas. Tais operações são duplicadas sem modificações na interface explícita, com duas exceções: operações de *factory* e *finder*.

4.9.6.1 Operações de *Factory*

Uma operação de *factory* é identificada na declaração de um *home* por meio da palavra-chave `factory`. Assim, a declaração, em IDL3:

```
home <home_name> manages <component_type> {
   factory <factory_operation_name> (<parameters>) raises (<exceptions>);
};
```

resulta na seguinte declaração, em IDL2, na interface explícita:

```
<component_type> <factory_operation_name> ( <parameters> )
   raises (Components::CreateFailure, <exceptions> );
```

Toda operação de *factory* devolve uma referência a uma nova instância de um componente. Para mais informações sobre a semântica de criação de objetos, veja [GHJV94].

4.9.6.2 Operações de *Finder*

Uma operação de *finder* é identificada na declaração de um *home* por meio da palavra-chave `finder`. Assim, a declaração, em IDL3:

```
home <home_name> manages <component_type> {
   finder <finder_operation_name> (<parameters>) raises (<exceptions>);
};
```

resulta na seguinte declaração, em IDL2, na interface explícita:

```
<component_type> <finder_operation_name> ( <parameters> )
   raises (Components::FinderFailure, <exceptions> );
```

A referência devolvida por uma operação de *finder* deve estar associada a uma instância de um componente pré-existente. A determinação de qual instância deve ser devolvida é de responsabilidade da implementação da operação. Para mais informações sobre a semântica de localização de objetos, veja [GHJV94].

4.9.7 Herança de *Homes*

A seguinte declaração, em IDL3:

```
home <home_name> : <base_home_name> manages <component_type> {
   <explicit_operations>
};
```

resulta na seguinte declaração, em IDL2, na interface explícita:

```
interface <home_name>Explicit : <base_home_name>Explicit {
   <equivalent_explicit_operations>
};
```

As declarações, em IDL3, de *homes* que admitem interfaces por herança, como:

```
home <home_name> : <base_home_name> supports <interface_name>
   manages <component_type> {
      <explicit_operations>
};
```

resultam, em IDL2, na interface explícita, em declarações de forma:

```
interface <home_name>Explicit : <base_home_name>Explicit,<interface_name> {
   <equivalent_explicit_operations>
};
```

Considerações sobre herança de *homes*:

- Um *home* derivado deve gerenciar o mesmo tipo de componente gerenciado pelo *home* base, ou derivado dele;
- Se a definição do *home* base incluir um tipo de chave primária, então a definição do *home* derivado deve incluir o mesmo tipo de chave primária, ou derivado dele. Caso a definição do *home* derivado omita o tipo de chave primária, implicitamente este será o mesmo do incluso na definição do *home* base;
- As operações da interface implícita não são herdadas pelo *home* derivado;
- As operações da interface `CCMHome` são herdadas por todas as interfaces equivalentes de *homes*, com ou sem chave primária;
- As operações da interface `KeylessCCMHome` são herdadas por todas as interfaces equivalentes de *homes* que não contenham chaves primárias.

4.9.8 Operações Ortodoxas e Heterodoxas

As operações contidas na definição de um *home* podem ser divididas em dois grupos:

- Operações Ortodoxas: são aquelas definidas pelo CCM. Aqui se enquadram as operações contidas na interface implícita do *home*, ou ainda nas interfaces `CCMHome` (seção 4.9.9) e `KeylessCCMHome` (seção 4.9.10). Sua implementação é responsabilidade do fornecedor do ORB. Essas operações, dado o modelo de herança descrito na seção anterior, não apresentam problemas relacionados ao polimorfismo;
- Operações Heterodoxas: são aquelas cuja implementação deve ser fornecida pelo programador, usuário do ORB. Estão contidas na interface explícita do *home*, constituindo-se de operações de *factory*, *finder*, operações definidas pelo usuário e atributos. Com relação ao polimorfismo, uma operação em um *home* base com chave primária, com parâmetro do tipo desta chave, possui, no *home* derivado, uma operação derivada onde o parâmetro pode ser do tipo da chave primária do *home* base ou derivado, a critério do programador. Não há, na especificação do CCM, definição da semântica para a igualdade polimórfica de chaves. Entretanto, para que o modelo seja

coerente, a semântica adotada para a comparação entre chaves deve ser a mesma para todas as operações heterodoxas no *home*.

Essa classificação de operações em ortodoxas ou heterodoxas serve para identificar claramente as operações definidas pelo CCM ou pelo usuário. Vale notar que este conceito é diferente do empregado na definição de interfaces implícitas e explícitas, não havendo relação de igualdade entre operações ortodoxas e interfaces implícitas, nem entre operações heterodoxas e interfaces explícitas. Por exemplo, operações da interface `CCMHome` são ortodoxas, pois foram definidas pelo CCM, mas estão na interface explícita dos *homes*.

4.9.9 A Interface `CCMHome`

Contém operações que se aplicam a qualquer *home*, independentemente da definição ou não de chave primária. Essa interface é herdada pela interface explícita de todos os *homes*. A definição, em IDL3, é:

```
module Components {
   typedef unsigned long FailureReason;
   exception CreateFailure { FailureReason reason; };
   exception FinderFailure { FailureReason reason; };
   exception RemoveFailure { FailureReason reason; };
   exception DuplicateKeyValue { };
   exception InvalidKey { };
   exception UnknownKeyValue { };
   interface CCMHome {
      CORBA::IRObject get_component_def();
      CORBA::IRObject get_home_def ();
      void remove_component ( in CCMObject comp)
         raises (RemoveFailure);
   };
};
```

A semântica das operações é:

- `get_component_def`: devolve uma referência a um objeto do tipo `CORBA::ComponentIR::ComponentDef`, descrevendo o tipo de componente associado ao *home*. Para linguagens fortemente tipadas, a referência devolvida deve ser estreitada para `CORBA::ComponentIR::ComponentDef` antes que possa ser utilizada;
- `get_home_def`: devolve uma referência a um objeto do tipo `CORBA::ComponentIR::HomeDef`, descrevendo o tipo do *home*. Para linguagens fortemente tipadas, a referência devolvida deve ser estreitada para `CORBA::ComponentIR::HomeDef` antes que possa ser utilizada;
- `remove_component`: destrói o componente cuja referência foi recebida como parâmetro. Chamadas subseqüentes a esta operação

resultam na exceção de sistema `OBJECT_NOT_EXIST`. Outras situações de erro devem disparar a exceção `RemoveFailure`.

A especificação do CCM não define quais devem ser os valores de `FailureReason` para as exceções `CreateFailure`, `FinderFailure` e `RemoveFailure`.

4.9.10 A Interface `KeylessCCMHome`

Aplica-se a *homes* que não contenham chave primária. É herdada pelas interfaces implícitas desses *homes*. A definição, em IDL3, é:

```
module Components {
   interface KeylessCCMHome {
      CCMObject create_component() raises (CreateFailure);
   };
};
```

A operação `create_component` cria uma nova instância do tipo de componente associado ao *home*. A implementação do *home* pode desabilitar esta operação, bastando não implementá-la. Neste caso, se a operação for invocada, a exceção de sistema `NO_IMPLEMENT` será disparada. Outras situações de erro devem disparar a exceção `CreateFailure`.

4.10 *Home Finders*

Homes gerenciam um tipo específico de componente. Assim, para utilizar componentes, o cliente deve obter referência a um determinado *home*. Para isso, CCM define a interface `HomeFinder`, semelhante, embora mais simples, à interface `CosLifeCicle::FactoryFinder` [LCS02], e que implementa um serviço de diretório para *component homes*.

Para que um cliente obtenha uma referência que dê suporte à interface `HomeFinder`, utiliza-se:

```
CORBA::ORB::resolve_initial_references("ComponentHomeFinder")
```

A interface `HomeFinder` é definida, em IDL3, como:

```
module Components {
   exception HomeNotFound { };
   interface HomeFinder {
      CCMHome find_home_by_component_type (in CORBA::RepositoryId comp_repid)
```

```
        raises (HomeNotFound);
      CCMHome find_home_by_home_type (in CORBA::RepositoryId home_repid)
        raises (HomeNotFound);
      CCMHome find_home_by_name (in string home_name)
        raises (HomeNotFound);
   };
};
```

Tal definição fornece ao cliente três opções para a localização de um *home*:

- `find_home_by_component_type`: devolve uma referência a um objeto *home* que gerencia componentes do tipo especificado pelo identificador em repositório recebido como parâmetro. Caso o *home* procurado não esteja correntemente registrado no repositório, a exceção `HomeNotFound` será disparada;
- `find_home_by_home_type`: devolve uma referência a um objeto *home* do tipo especificado pelo identificador em repositório recebido como parâmetro. Caso o *home* procurado não esteja correntemente registrado no repositório, a exceção `HomeNotFound` será disparada;
- `find_home_by_name`: devolve uma referência a um objeto *home* associado ao nome recebido como parâmetro. Esse nome deve seguir o formato definido na especificação do Serviço de Nomes [INS02, na seção 2.4, "*Stringified Names*"]. A implementação desta operação pode manter múltiplas associações de *homes* a um determinado nome e, neste caso, é responsável por decidir a qual deles se aplica a referência devolvida. Isso é útil, por exemplo, para aumentar a escalabilidade da solução, fornecendo a possibilidade de devolver uma referência mais próxima ao cliente. Caso o nome procurado não esteja mapeado a um objeto *home* registrado no *home finder*, a exceção `HomeNotFound` será disparada.

Com relação à referência devolvida pelas operações descritas acima, valem as seguintes considerações:

- Uma vez que as operações devolvem referência do tipo `CCMHome`, esta deve ser estreitada (*narrowed*) para o tipo específico do *home* em questão antes de ser utilizada;
- O CCM garante apenas que a referência devolvida dá suporte à interface `CCMHome`. Quaisquer outras suposições feitas pelo cliente, como intuitivamente associar determinadas operações a um nome, não são garantidas;
- A interface `HomeFinder` é propositalmente simples. Caso o cliente necessite ser mais específico quanto ao *home* procurado, pode-se utilizar o serviço *Trading* [TRAD00] ou qualquer outro mecanismo que permita buscas para a seleção do *home* apropriado.

4.11 Configuração de Componentes

Uma característica desejável, em se tratando de componentes, é que estes sejam flexíveis e genéricos o suficiente para que possam ser utilizados por aplicações diversas. Com esse intuito, o CCM provê recursos tais como delegação, implementada através de portas como os receptáculos, e configurabilidade.

No que diz respeito a configuração de componentes, o CCM prevê as seguintes características:

- Capacidade de definir atributos para um tipo de componente, a serem utilizados no processo de configuração de uma instância do componente. É intenção do CCM que os atributos sejam utilizados durante a inicialização de uma instância do componente para estabelecimento de seu estado inicial, através de operações de *factory* ou de ferramentas de implantação. Embora não haja restrições quanto ao uso ou visibilidade dos atributos, acredita-se [CCM02] que as aplicações clientes usuárias das instâncias de um componente não se interessem por eles, uma vez que o componente esteja configurado;
- Capacidade de armazenar configurações em arquivos (seção 7.1.1.1, “Arquivo Descritor de Propriedades (.cpf)”) a serem utilizados no momento da implantação. Isso garante independência da geração do instalador;
- Capacidade de definir a configuração sem a necessidade de instanciar o componente.

O CCM permite, nas interfaces de um componente, uma distinção entre aquelas para serem utilizadas preferencialmente na configuração de componentes (interfaces de configuração) e outras a serem utilizadas preferencialmente por aplicações clientes em sua operação normal (interfaces operacionais). Na prática, todavia, essa distinção pode não ser desejada. Por esse motivo não é obrigatória.

As interfaces operacionais deveriam ser compostas por interfaces admitidas por herança e pelas facetas. Já as interfaces de configuração englobam os outros tipos de portas (receptáculos, fontes e consumidores de eventos) e os atributos.

É fornecido, pelo CCM, um mecanismo para definir as fases de configuração e operacional no ciclo de vida de um componente. Há ainda, como desabilitar determinadas interfaces em função da fase em que o componente se encontra.

4.11.1 Fases de Configuração e Operacional

Quando houver necessidade de uma fase de configuração explícita antes que se possam utilizar operações funcionais, o ciclo de vida do componente é dividido em duas fases mutuamente exclusivas: a de configuração e a operacional.

A existência ou não da fase de configuração é definida por meio do elemento `configurationcomplete` no arquivo descritor do componente (seção 7.1.1.2).

Havendo a fase de configuração, a operação `configuration_complete`, herdada de `Components::CCMObject`, deve ser invocada pela aplicação responsável pela configuração do componente. Essa operação somente termina quando a instância do componente tiver concluído suas tarefas de configuração, estando pronta para receber requisições dos clientes, ou quando a exceção `InvalidConfiguration` for disparada, indicando que o componente não se encontrava na fase de configuração quando da chamada da operação.

4.11.2 Configuração com Atributos

O CCM propõe, como mecanismo primário para a configuração de componentes, o uso de atributos. Ao conjunto formado por nomes de atributos e seus valores denomina-se configuração com atributos. Trata-se de uma descrição das operações de atribuição de valores a atributos a serem realizadas no processo de configuração.

CCM define, ainda, um conjunto de mecanismos básicos para a configuração de componentes, que podem ser empregados por operações de *factory*, por ferramentas de implantação ou mesmo pelas aplicações clientes que solicitarem a criação de componentes.

As próximas duas seções apresentam as interfaces envolvidas na representação e uso de uma configuração com atributos, e como configurar componentes por meio de operações de *factory*.

4.11.2.1 Configuradores de Atributos

Um configurador é um objeto que encapsula uma configuração com atributos que possa ser aplicada a instâncias de um tipo de componente. Apesar de possuírem a capacidade de utilizar quaisquer operações habilitadas durante a fase de configuração, os configuradores foram idealizados primariamente para utilizar operações de atribuição de valores a atributos de uma instância de um componente.

Um configurador é definido através das seguintes interfaces, em IDL3:

```
module Components {
   exception WrongComponentType { };

   valuetype ConfigValue {
      public FeatureName name;
      public any value;
   };
   typedef sequence<ConfigValue> ConfigValues;

   interface Configurator {
      void configure (in CCMObject comp) raises (WrongComponentType);
   };

   interface StandardConfigurator : Configurator {
      void set_configuration (in ConfigValues descr);
   };
};
```

A utilização das interfaces acima obedece à semântica:

- Definir um conjunto de nomes de atributos e seus valores, através da seqüência `ConfigValues`;
- Com a operação `set_configuration`, estabelecer como corrente a configuração definida;
- Associar, com a operação `configure`, a configuração corrente a uma instância de componente. Serão invocadas as operações de atribuição para os atributos presentes na seqüência `ConfigValues`. Caso o tipo de componente recebido como parâmetro seja diferente do esperado, a exceção `WrongComponentType` será disparada.

4.11.2.2 Configuração Baseada em Operações de *Factory*

Operações de *factory*, que estão presentes no *home* do componente, podem ser utilizadas no processo de configuração de instâncias de componentes de diversas maneiras:

- Uma operação de *factory* pode ser implementada de modo a configurar a instância explicitamente. Por exemplo, a operação pode receber, como parâmetro, um conjunto de valores de configuração, possivelmente na forma de seqüência do tipo `Components::ConfigValues`, e então realizar a configuração;
- Operações de *factory* podem invocar a operação `configuration_complete` explicitamente, ou através de solicitação de uma aplicação de implantação;
- Uma operação de *factory* pode utilizar valores de um configurador fornecido por uma aplicação de implantação. Para isso, o *home* pode dar suporte à interface `HomeConfiguration`, definida como:

```
module Components {
   interface HomeConfiguration : CCMHome {
      void set_configurator (in Configurator cfg);
      void set_configuration_values (in ConfigValues config);
      void complete_component_configuration (in boolean b);
      void disable_home_configuration();
   };
};
```

A semântica das operações é:

- `set_configurator`: associa um configurador ao *home*, a ser utilizado na criação de instâncias do componente que este gerencia;
- `set_configuration_values`: armazena uma configuração com atributos;
- `complete_component_configuration`: determina, em função do parâmetro recebido, se operações de *factory* no *home* chamarão a operação `configuration_complete` para cada instância criada do componente;
- `disable_home_configuration`: desabilita futuras chamadas a operações da interface `HomeConfiguration` que, se então invocadas, ocasionarão o disparo da exceção `BAD_INV_ORDER`. Pode ser interpretada como um delimitador entre as fases de configuração e operacional para *homes*, analogamente à operação `configuration_complete` para os componentes.

O CIF define implementações padrão para operações de *factory*, geradas automaticamente, e que se comportam como acima descrito. Entretanto, os implementadores de componentes podem optar pelo desenvolvimento de operações personalizadas, dando ou não suporte à interface `HomeConfiguration`.

Caso não sejam definidos mecanismos de configuração de atributos, será realizada apenas a inicialização prevista no construtor do servente do componente.

Capítulo 5

Contêineres

Conforme descrito na seção 3.1, “Exemplo de Utilização”, os componentes são empacotados em DLLs e executados em servidores de componentes. As implementações dos componentes dependem do POA para encaminhar requisições de clientes para os serventes.

Os componentes não precisam saber como tratar problemas como a criação de hierarquia de POAs e localizar serviços do CCM. Para isso foram definidos os contêineres, que são a estrutura para integrar, em tempo de execução, o componente aos serviços de CORBA, tais como transações, notificação, persistência e segurança. Os contêineres possuem as seguintes funcionalidades:

- Ativação/desativação de implementações de componentes, preservando recursos (como memória);
- Fornecimento de camada de adaptação com os serviços de transação, persistência, segurança e notificação;
- Fornecimento de camada de adaptação para *callbacks*;
- Gerenciamento de políticas do POA.

5.1 Arquitetura do Modelo de Programação de Contêineres

Descreve a relação entre o contêiner e os demais elementos do CCM. A arquitetura do modelo de programação de contêineres pode ser ilustrada pela Figura 5.1.

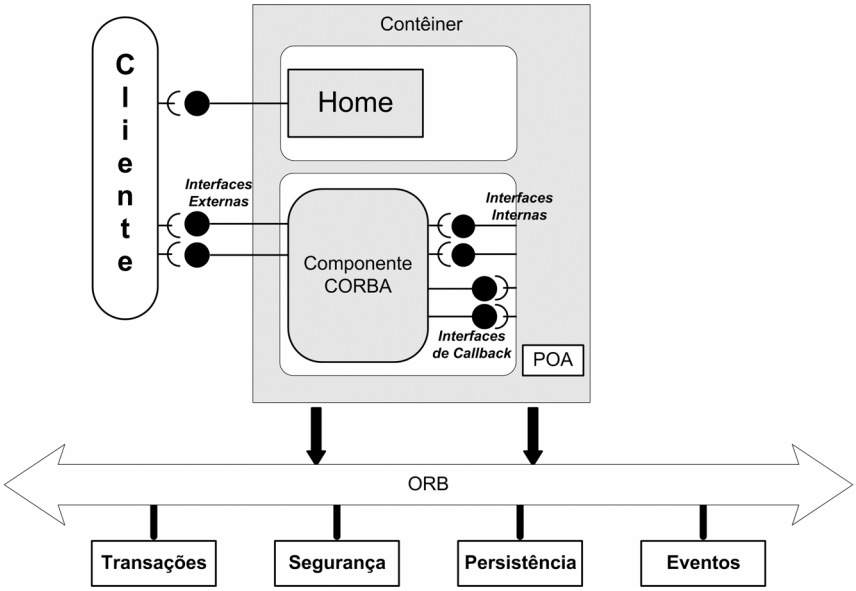

Figura 5.1: arquitetura do modelo de programação de contêineres.
O componente é executado no contexto do contêiner. Note as interfaces: externas, disponibilizadas pelo componente e utilizadas pelo cliente; internas, disponibilizadas pelo contêiner e utilizadas pelo componente; e de *callback*, disponibilizadas pelo componente e utilizadas pelo contêiner. Adaptado de [CCM02].

Fazem parte deste modelo de programação:

- APIs externas, para comunicação entre o cliente e o componente, correspondendo às interfaces disponíveis ao cliente, definidas em IDL e armazenadas no repositório de interfaces;
- APIs do contêiner, para comunicação entre contêiner e componente, composto pelas interfaces internas e de *callback*, utilizadas pelo desenvolvedor do componente;
- Modelo de utilização CORBA, que especifica como o contêiner, o POA e os serviços de CORBA se integram;
- Categorias de componentes, que são o conjunto das combinações válidas entre os três elementos anteriores e são especificadas em arquivo CIDL (detalhado no Capítulo 6).

5.1.1 Tipos de APIs Externas

São constituídos por um conjunto de interfaces expostas pelo componente aos clientes, determinando um contrato de utilização. Essas interfaces são divididas em dois grupos:

- Interface `Home`: descrita na seção 4.9, "*Homes*";
- Interfaces de aplicação: definidas em IDL pelo desenvolvedor do componente.

5.1.2 Tipos de APIs dos Contêineres

Constituídos por conjuntos de interfaces internas, para comunicação entre o componente e o contêiner, e de *callbacks*, para comunicação entre o contêiner e o componente.

CCM define dois tipos de APIs dos contêineres:

- Sessão: para componentes que utilizem referências transientes;
- Entidade: para componentes que utilizem referências persistentes.

O tipo de API do contêiner é implicitamente selecionado ao se escolher a categoria do componente (seção 5.1.4, “Categorias de Componentes”).

As interfaces que compõem as APIs dos contêineres podem ser agrupadas em três conjuntos: específicas para sessão, específicas para entidade e comum aos dois. Estas interfaces são descritas nas seções 5.2.3 e 5.2.4.

5.1.3 Modelo de Utilização CORBA

Um dos benefícios do CCM é facilitar a configuração das políticas do POA. O modelo de utilização CORBA, que faz parte da especificação do CCM, prevê as combinações válidas entre duas dessas políticas: o tipo de referências a objetos, transientes ou persistentes, e a cardinalidade do mapeamento entre serventes e referências a objetos (ObjectID), 1:1 ou 1:N. Existem três modelos possíveis:

- sem estado (*stateless*): usa referências transientes a objetos com um servente que atende a qualquer ObjectId;
- conversacional (*conversational*): usa referências transientes com servente dedicado a um ObjectId específico;
- durável: usa referências persistentes com servente dedicado a um ObjectId específico.

Cada modelo de utilização CORBA define ainda a integração com os serviços de CORBA.

Os modelos podem ser resumidos conforme a Tabela 5.1.

Modelo de Utilização CORBA	Políticas do POA		Serviços de CORBA		
	Tipo de Referências a Objetos	Mapeamento Servente: ObjectID	Persistência	Transações	Notificação
Sem estado	Transiente	1:N	Não utiliza	Pode utilizar mas não pode fazer parte da transação corrente	Transacional ou não
Conversacional	Transiente	1:1	Não utiliza	Pode utilizar mas não pode fazer parte da transação corrente	Transacional ou não
Durável	Persistente	1:1	Pode utilizar[6]	Pode utilizar e pode fazer parte da transação corrente	Transacional ou não
(Inválido)	Persistente	1:N	-	-	-

Tabela 5.1: modelos de utilização CORBA.
Vale notar que a última combinação, referências persistentes com mapeamento 1:N, não faz sentido.

Cada componente é utilizado de acordo com um e somente um dos modelos acima. A escolha do modelo a utilizar é implicitamente realizada ao selecionar a categoria do componente (seção 5.1.4, “Categorias de Componentes“).

5.1.3.1 Tipos de Referências a Objetos

As referências a objetos CORBA podem ser de dois tipos: transientes ou persistentes, conforme definido pela política `LifespanPolicy` do POA [HV99, seção 11.4.1, “*CORBA Object Life Span*”].

Referências transientes são utilizadas para objetos temporários, o que pode ser estendido para componentes.

Já referências persistentes devem ser empregadas para objetos (e também para componentes) que necessitem armazenar seu estado em meio durável, como bancos de dados, por exemplo.

[6] Neste caso o desenvolvedor pode optar por outro mecanismo de persistência que não o Serviço de Persistência CORBA (PSS).

5.1.3.2 Mapeamento entre Referências e Serventes

O modelo de utilização CORBA permite o uso de objetos sem estado (*stateless*). Quando isso ocorre, o CCM aplica o mapeamento entre serventes e ObjectID com cardinalidade 1:N, o que aumenta a escalabilidade da solução, conforme ilustrado na Figura 5.2[7]. Quando o componente possui estado, seja transiente ou persistente, o mapeamento é 1:1, conforme a Figura 5.3.

O mapeamento entre referências e serventes pode ser traduzido como a política `IdUniquenessPolicy` do POA [HV99, seção 11.4.3, "*Mapping Objects to Servants*"].

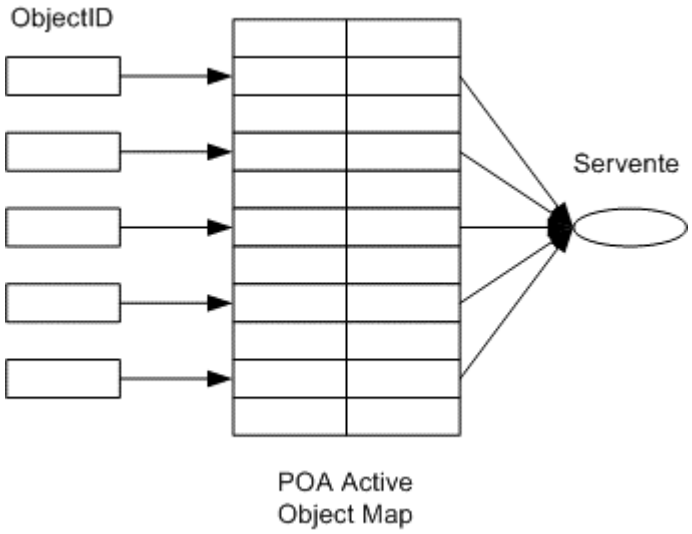

Figura 5.2: mapeamento de 1 servente para múltiplos identificadores de objeto

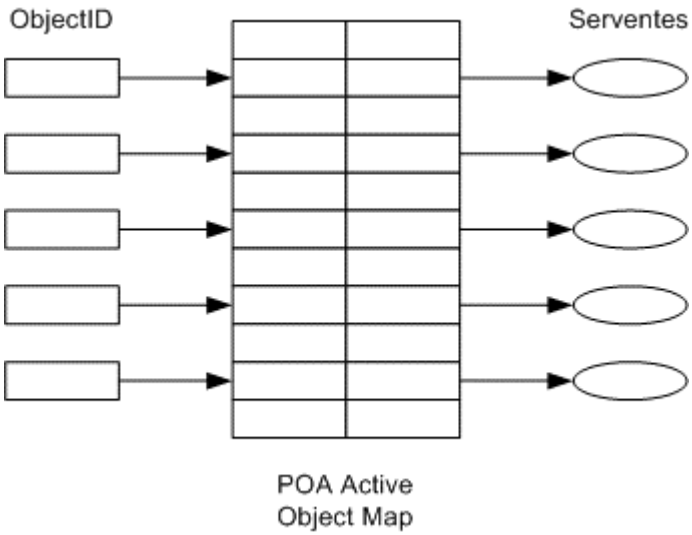

Figura 5.3: mapeamento de 1 servente para cada identificador de objeto

[7] As figuras apresentadas consideram a utilização de POA *Active Object Map* apenas como ilustração. Poderia ter sido utilizado um gerenciador de serventes diferente.

5.1.3.3 Modelos de Execução (utilização de *threads*)

A política de execução do POA é estabelecida no descritor de componente CORBA (seção 7.1.1.2, “Arquivo Descritor de Componente (.ccd)”), que faz parte do modelo de empacotamento de componentes, assunto do Capítulo 7. Um componente CORBA pode ser executado pelo contêiner em dois modos: uma única *thread* de execução ou múltiplas *threads*. Esses modos são definidos através dos valores `serialize` ou `multithread`, nesta ordem, do elemento `threading` do descritor de componentes.

5.1.4 Categorias de Componentes

A relação entre o modelo de utilização CORBA, o tipo de API do contêiner e a presença de *home* com chave primária é conceituada como Categoria de Componente. O objetivo deste conceito é facilitar a tarefa do desenvolvedor de componentes, evitando que este tenha de fazer um conjunto de configurações. Assim, ao estabelecer a categoria do componente, o desenvolvedor implicitamente define políticas do POA, além de selecionar o conjunto de APIs do contêiner mais adequado ao componente desejado.

A definição da categoria do componente é realizada empregando-se o arcabouço para implementação de componentes (Capítulo 6, “CCM Implementation Framework – CIF”), na linguagem CIDL (seção 6.1.2, “Estrutura Básica de Composições”).

A Tabela 5.2 sintetiza as quatro categorias de componentes previstas na especificação do CCM.

Categoria de componente	Modelo de Utilização CORBA	Tipo de API do Contêiner	Interface `Home` com chave primária	Exemplo
Serviço	Sem estado	Sessão	Não	*Wrappers* para aplic. procedurais legadas
Sessão	Conversacional	Sessão	Não	Iteradores
Processo	Durável	Entidade	Não	Regras de negócio. Ex.: carrinho de compras
Entidade	Durável	Entidade	Sim	Peças num inventário

Tabela 5.2: categorias de componentes.
Estão representadas as relações com o modelo de utilização CORBA, o tipo de API do contêiner e a presença de interface `Home` com chave primária. Baseado em [CCM02].

Além das quatro categorias previstas pelo CCM, há a possibilidade do desenvolvedor escolher outras combinações de políticas do POA. Quando isso é feito, a categoria do componente é vazia (*empty*).

5.1.4.1 Serviço

Um componente de serviço pode ser visto como uma biblioteca de funções que não necessita de manutenção de estado. Um componente que efetue cálculos, como um conjunto de funções estatísticas, pode ser implementado como serviço.

5.1.4.2 Sessão

Componentes de sessão mantêm estado, porém de modo transiente. Um componente de segurança que armazene a identificação de um usuário durante o tempo em que este utilizar determinado sistema exemplifica esta categoria de componente. Quando a aplicação for encerrada, o componente e seu estado serão destruídos.

5.1.4.3 Processo

Componentes de processo armazenam estado de modo persistente. Entretanto, o cliente não recebe um identificador do componente. Assim, o cliente não possui acesso direto ao componente através de uma chave. Componentes que representam processos de negócio, como um carrinho de compras, ilustram esta categoria de componente.

5.1.4.4 Entidade

Estes componentes, assim como os componentes de processo, possuem estado persistente. Contudo, representam entidades como pessoas, contas correntes ou peças. Assim sendo, cada instância deste componente pode ser acessado diretamente pelo cliente através de uma chave.

5.1.4.5 Vazio

Uma das principais finalidades do CCM é facilitar o trabalho do desenvolvedor, fornecendo a este recursos que evitem gasto excessivo de tempo com configuração de políticas do POA e gerenciamento de ciclo de vida de objetos. Todavia, existem casos em que as combinações de configuração de políticas de POA previstas pelo CCM não atendem a necessidades específicas.

Para estas situações, o desenvolvedor de componentes pode configurar cada política do POA, através do descritor de componentes CORBA, conforme descrito na seção 7.1.1.2, “Arquivo Descritor de Componente (.ccd)”. Um exemplo deste caso é o uso de persistência para referências transientes, para balanceamento de carga. Essa combinação não é contemplada pelas demais categorias de componentes.

5.2 Considerações sobre a Programação de Servidores

O uso de componentes, para ser escalável, exige cuidados no gerenciamento do ciclo de vida, assunto da seção 5.2.1, “Ciclo de Vida de Componentes”.

Os componentes CORBA são utilizados pelos clientes através de conjuntos de interfaces específicas. Basicamente, estes conjuntos variam em função de dois fatores: o tipo do componente (seção 4.1) e o tipo de APIs do contêiner. Essas interfaces são descritas nas seções 5.2.3, “Interfaces de Programação para Componentes Servidores Básicos” e 5.2.4, “Interfaces de Programação para Componentes Servidores Estendidos”.

5.2.1 Ciclo de Vida de Componentes

5.2.1.1 Criação de Instâncias de Componentes

A criação de instâncias de um componente é realizada pelo *home*, por meio de operações de *factory*, em geral a partir do cliente. Pode, entretanto, ser feita através da implementação do componente. Um *factory* padrão é gerado automaticamente ao se compilar a definição em IDL do componente. Outras implementações podem ser realizadas pelo desenvolvedor, dependendo de necessidades específicas, como a de enviar parâmetros durante a criação de instância do componente.

5.2.1.2 Ativação de Componentes

Os componentes são ativados automaticamente pelo POA, que utiliza informações constantes do descritor do componente (seção 7.1.1.2, “Arquivo Descritor de Componente (.ccd)”). Os clientes fazem chamadas a referências previamente exportadas. As requisições são roteadas pelo ORB ao POA que criou as referências e este, por sua vez, repassa ao contêiner. Esse mecanismo é o que permite ao contêiner controlar a ativação e desativação do componente, efetuar as chamadas de *callback*, do contêiner ao componente, e aplicar as políticas definidas no descritor do componente.

A chamada a qualquer operação de um componente faz com que o segmento (seção 6.1.6.1, “Executores Monolíticos e Segmentados”) onde esta foi definida seja ativado, o que conseqüentemente consome recursos, como memória e UCP. No caso das operações da interface `Navigation`, por exemplo, uma boa implementação do CCM poderia interceptar as chamadas e não ativar o componente [CCM02, nota na página 4-31].

5.2.1.3 Gerenciamento do Ciclo de Vida

O gerenciamento do tempo de vida dos serventes é feito através de políticas que controlam o momento de ativação/desativação dos componentes. O POA utiliza os serventes para enviar requisições ao objeto alvo, a partir do ObjectID, presente na chave do objeto (*object key*). O modelo de programação de servidores permite que o desenvolvedor do componente faça um conjunto de escolhas que afetam diretamente o modo como o ciclo de vida do componente é gerenciado. Essas escolhas são:

- A categoria do componente que, implicitamente, define o modelo de utilização, o tipo de API do contêiner e a presença de chave primária;
- A política de ciclo de vida dos serventes, definida no descritor de componente (seção 7.1.1.2, “Arquivo Descritor de Componente (.ccd)”), que pode ser:
    - Método: ativação/desativação a cada chamada de método, limitando o uso de memória ao tempo de duração da operação, mas acrescentando o custo de ativação e desativação do componente;
    - Transação: ativação/desativação a cada transação. Memória permanece alocada durante a transação;
    - Componente: o contêiner ativa o componente quando for feita a primeira chamada a alguma de suas operações, e desativa quando explicitamente requisitado pela aplicação, desalocando a memória utilizada pelo componente;
    - Contêiner: o componente será ativado quando for feita a primeira chamada a alguma de suas operações e, ao final da execução da mesma, será desativado. Entretanto, a memória permanecerá alocada até que o contêiner decida desalocá-la.

O modelo de programação de servidores também presume que as rotinas de *callback*, quando presentes, devem ser implementadas.

A Tabela 5.3 relaciona a categoria do componente com as políticas de ciclo de vida dos serventes.

Categoria de componente	Políticas de ciclo de vida dos serventes válidas
Serviço	Método
Sessão	Método, Transação, Componente e Contêiner
Processo	Método, Transação, Componente e Contêiner
Entidade	Método, Transação, Componente e Contêiner

Tabela 5.3: categorias de componentes e sua relação com as políticas válidas para o ciclo de vida dos serventes. Baseado em [CCM02].

5.2.2 Integração com Serviços de CORBA

5.2.2.1 Transações

Os Componentes CORBA permitem dois tipos de gerenciamento de transações: gerenciamento pelo componente (SMT, *self-managed transactions*) ou pelo contêiner (CMT, *container-managed transactions*).

O gerenciamento de transações pelo componente deve ser implementado pelo desenvolvedor, utilizando mecanismos como o serviço de transações CORBA [TS02] ou a interface `UserTransaction` do contêiner (seção 5.2.3.1, “Interfaces Comuns aos Tipos de APIs dos Contêineres”).

O gerenciamento de transações pelo contêiner implica na definição da política de transações a empregar através do descritor do componente (seção 7.1.1.2, “Arquivo Descritor de Componente (.ccd)”). Essas políticas fornecem informações ao contêiner sobre como este deve conduzir o comportamento transacional do componente. As políticas previstas pelo CCM são:

- `not_supported`: o componente não utiliza transações, mesmo que a aplicação que o chamou esteja no contexto de uma transação, caso em que esta será suspensa;
- `never`: o componente não permite o uso de transações. Se a aplicação que o chamou estiver no contexto de uma transação, a exceção `INVALID_TRANSACTION` será disparada;
- `supported`: o componente utiliza a transação da aplicação que o chamou, se houver. Caso contrário, o componente não utiliza transações;
- `required`: o componente exige uma transação. Se a aplicação que o chamou estiver no contexto de uma transação, esta será utilizada. Caso contrário, nova transação será iniciada;

- `requires_new`: o componente exige uma transação nova. Se a aplicação que o chamou estiver no contexto de uma transação, esta será suspensa;
- `mandatory`: o cliente exige que a aplicação que o chamou esteja no contexto de uma transação e a utiliza. Caso isso não ocorra, a exceção `TRANSACTION_REQUIRED` será disparada.

O mecanismo de transação utilizado pelos Componentes CORBA e pelo Serviço de Transações CORBA é o de efetivação bifásica (*two-phase commit*).

5.2.2.2 Segurança

O contêiner, baseado nas políticas de segurança configuradas no descritor do componente, utiliza o Serviço de Segurança CORBA [SS02] para verificar as credenciais ativas durante as chamadas de operações.

As permissões de acesso podem ser estabelecidas para cada porta de um componente, bem como para seu *home*. Desse modo, a utilização das portas de um componente pode ser personalizada.

O transporte de credenciais entre sistemas, comum em aplicações distribuídas, só é possível através do protocolo SECIOP [SS02].

5.2.2.3 Eventos e Notificação

Conforme descrito na seção 4.7, “Eventos”, os componentes CORBA se utilizam de um subconjunto do Serviço de Notificação CORBA [NSS02] para emitir e consumir eventos. Políticas de transação e segurança podem ser aplicadas a partir do descritor do componente (seção 7.1.1.2, “Arquivo Descritor de Componente (.ccd)”).

O contêiner é responsável por mapear os eventos dos componentes ao serviço de notificação.

Políticas de Transação para Eventos

O descritor do componente pode definir políticas a serem aplicadas a cada porta de evento, seja fonte ou consumidora. As possíveis opções estão descritas a seguir, comparadas com valores da seção 5.2.2.1, “Transações”:

- `normal`: semelhante ao valor `not_supported`, ou seja, o evento estará fora do escopo de uma transação. Se houver uma transação ativa, esta será suspensa até o término do envio/consumo do evento;

- `default`: semelhante ao valor `supported`. Assim, o evento estará no escopo de uma transação apenas se já houver transação ativa;
- `transaction`: semelhante ao valor `required`, isto é, o evento será emitido/consumido no escopo de uma transação. Desse modo, se houver uma transação ativa, esta será utilizada. Caso contrário, uma nova transação será criada.

Políticas de Segurança para Eventos

O controle de acesso a portas de eventos (fontes ou consumidoras), é baseado em papéis que devem ser associados a elas através de ACLs (*access control lists*, que são relações de papéis com permissão de acesso ao recurso, no caso as portas). Estas políticas de segurança são então utilizadas pelo Serviço de Segurança CORBA [SS02] para restringir acesso aos eventos.

5.2.2.4 Persistência

Os componentes de processo ou entidade, ou seja, que dêem suporte à API do contêiner do tipo entidade, necessitam de mecanismos de persistência para armazenamento de estado. Neste ponto, faz-se mister apresentar duas definições:

- Mecanismos de Persistência: tratam do ato de persistir o estado, isto é, expressam como a persistência será implementada. Há duas formas possíveis: Serviço de Persistência de Estado CORBA ou implementação definida pelo desenvolvedor. No primeiro caso, as implementações de operações do arcabouço de persistência, ou seja, de *factory*, *finder* e algumas de *callback* podem ser geradas automaticamente. Já no segundo caso, tais operações devem ser supridas pelo desenvolvedor. Veja mais detalhes na seção 6.1.3, “Composições com Armazenagem Gerenciada”;
- Gerenciamento de Persistência: trata do acionamento dos mecanismos de persistência, podendo ser realizado pelo contêiner (CMP, *container-managed persistence*) ou pelo componente (SMP, *self-managed persistence*). No gerenciamento pelo contêiner, o desenvolvedor apenas define o estado a persistir e o contêiner se incumbe de acionar o mecanismo de persistência para salvar e recuperar o estado. No gerenciamento pelo componente, o desenvolvedor deve implementar as interfaces de *callback* (seções 5.2.3.3 e 5.2.4.3), a serem invocadas pelo contêiner.

A Tabela 5.4 ilustra as possíveis combinações entre os mecanismos e o gerenciamento de persistência, apresentando os responsáveis pela

implementação das classes do arcabouço de persistência e as interfaces de *callback*.

Mecanismo de Persistência	Gerenciamento de Persistência	Classes de Persistência	Interfaces de *callback*
PSS	Contêiner (CMP)	Geradas	Geradas
PSS	Componente (SMP)	Geradas	Implementadas pelo componente
Desenvolvedor	Contêiner (CMP)	Implementadas pelo componente	Geradas
Desenvolvedor	Componente (SMP)	Implementadas pelo componente	Implementadas pelo componente

Tabela 5.4: relação entre os mecanismos e gerenciamento de persistência.
As classes do arcabouço de persistência são geradas automaticamente quando o mecanismo é o Serviço de Persistência de Estado CORBA (PSS). Do mesmo modo, as interfaces de *callback* são geradas automaticamente quando o gerenciamento de persistência é realizado pelo contêiner (CMP). Baseado em [CCM02].

5.2.3 Interfaces de Programação para Componentes Servidores Básicos

Conforme visto na seção 4.1, “Tipos de Componentes”, os componentes básicos podem ser utilizados por clientes que estejam cientes ou não de sua interação com um componente CORBA. Para isso, os componentes básicos possuem recursos limitados, não podendo possuir portas, o que faz com que sejam vistos pelo cliente como objetos CORBA ao invés de componentes. Além disso, a integração com EJB só é possível se o componente CORBA for básico.

Nesta seção serão apresentadas as interfaces admitidas pelos componentes básicos. Algumas dessas interfaces independem do tipo de API do contêiner. Outras, contudo, são específicas para cada tipo de API, sessão ou entidade.

5.2.3.1 Interfaces Comuns aos Tipos de APIs dos Contêineres

A Interface `CCMContext`

É uma interface interna, que fornece ao componente informações sobre o contexto em que este é executado e acesso aos serviços disponibilizados pelo contêiner. Isso garante que o componente obtenha todas as referências de que possa precisar para implementar seu comportamento. Sua definição, em IDL3, é:

```
typedef SecurityLevel2::Credentials Principal;
exception IllegalState { };

local interface CCMContext {
   Principal get_caller_principal();
   CCMHome get_CCM_home();
   boolean get_rollback_only() raises (IllegalState);
   Transaction::UserTransaction get_user_transaction() raises (IllegalState);
   boolean is_caller_in_role (in string role);
   void set_rollback_only() raises (IllegalState);
};
```

A semântica das operações é:

- `get_caller_principal`: devolve as credenciais de segurança associadas ao chamador;
- `get_CCM_home`: devolve referência ao *home* associado ao componente;
- `get_rollback_only`: devolve se a transação corrente está marcada para ser desfeita (*rollback*);
- `get_user_transaction`: devolve uma referência à transação corrente. Esta operação deve ser utilizada apenas para transações gerenciadas pelo componente (SMT). Se a forma de gerenciamento for CMT (contêiner), a exceção `IllegalState` será disparada;
- `is_caller_in_role`: compara as credenciais de segurança do chamador da operação, isto é, o cliente, com as do papel `role`. Por exemplo, esta operação pode ser utilizada para testar regras de negócio como: "gerentes podem aprovar solicitações de até R$ 100.000,00";
- `set_rollback_only`: marca a transação corrente para ser desfeita (*rollback*). Se não houver transação, a exceção `IllegalState` será disparada.

A Interface `Home`

É uma interface externa que admite operações de *factory* e *finder*, conforme a seção 4.9, "*Homes*".

A Interface `UserTransaction`

Componentes que gerenciam transações (SMT) podem fazê-lo através do Serviço de Transações CORBA ou através de outra API, definida pelo programador. Caso utilize o Serviço de Transações CORBA, o contêiner implementa e disponibiliza a interface `UserTransaction` para ser utilizada

pelo componente sendo, portanto, uma interface interna. A sintaxe desta interface, em IDL3, é:

```
typedef sequence<octet> TranToken;
exception NoTransaction { };
exception NotSupported { };
exception SystemError { };
exception RollbackError { };
exception HeuristicMixed { };
exception HeuristicRollback { };
exception Security { };
exception InvalidToken { };
enum Status {ACTIVE, MARKED_ROLLBACK, PREPARED, COMMITTED, ROLLED_BACK,
   NO_TRANSACTION, PREPARING, COMMITTING, ROLLING_BACK};

local interface UserTransaction {
   void begin () raises (NotSupported, SystemError);
   void commit () raises (RollbackError , NoTransaction,
      HeuristicMixed, HeuristicRollback,Security, SystemError);
   void rollback () raises (NoTransaction, Security, SystemError);
   void set_rollback_only () raises (NoTransaction, SystemError);
   Status get_status() raises (SystemError);
   void set_timeout (in long to) raises (SystemError);
   TranToken suspend () raises (NoTransaction, SystemError);
   void resume (in TranToken txtoken) raises (InvalidToken, SystemError);
};
```

A semântica das operações é:

- `begin`: começa uma transação;
- `commit`: encerra uma transação com sucesso;
- `rollback`: encerra uma transação com insucesso;
- `set_rollback_only`: marca uma transação para ser desfeita;
- `get_status`: devolve a situação da transação corrente;
- `set_timeout`: estabelece a duração máxima, em segundos, da transação corrente;
- `suspend`: desconecta a transação corrente da *thread* corrente, ou seja, suspende a transação;
- `resume`: reconecta a transação corrente à *thread* corrente, dando continuidade à transação.

A Interface `EnterpriseComponent`

É uma interface de *callback*, que não define operações. Todo componente CORBA deve ter uma interface derivada desta, para ser acionada a partir do contêiner. Sua sintaxe, em IDL3, é:

```
local interface EnterpriseComponent{ };
```

5.2.3.2 Interfaces para o Tipo de API de Contêiner Sessão

A Interface `SessionContext`

É uma interface interna, derivada de `CCMContext`, e fornece a uma instância do componente acesso aos serviços disponibilizados pelo contêiner. Sua sintaxe, em IDL3, é:

```
exception IllegalState { };

local interface SessionContext : CCMContext {
    Object get_CCM_object() raises (IllegalState);
};
```

A operação `get_CCM_object` deve ser utilizada no escopo de uma operação de *callback* (caso contrário, a exceção `IllegalState` será disparada), e devolve uma referência à instância do componente que o contêiner utilizou para invocar a operação de *callback*. Para componentes básicos, é a referência à interface equivalente. Para os estendidos (onde esta interface é estendida pela interface `Session2Context`, conforme a seção 5.2.4.2), é referência a uma faceta.

A Interface `SessionComponent`

É uma interface de *callback*, utilizada pelo contêiner como parte do gerenciamento do ciclo de vida dos serventes. Sua sintaxe, em IDL3, é:

```
enum CCMExceptionReason {SYSTEM_ERROR, CREATE_ERROR, REMOVE_ERROR,
   DUPLICATE_KEY, FIND_ERROR, OBJECT_NOT_FOUND, NO_SUCH_ENTITY};
exception CCMException {CCMExceptionReason reason;};

local interface SessionComponent : EnterpriseComponent {
   void set_session_context ( in SessionContext ctx) raises (CCMException);
   void ccm_activate() raises (CCMException);
   void ccm_passivate() raises (CCMException);
   void ccm_remove () raises (CCMException);
};
```

A semântica das operações é:

- `set_session_context`: após criar uma instância do componente, mas fora do contexto de transações, o contêiner chama esta operação para informar ao componente o contexto em que este é executado;
- `ccm_activate`: chamada pelo contêiner para informar ao componente que este foi ativado. A configuração/inicialização do componente deve ter sido realizada antes que esta operação seja invocada (seção 4.11, “Configuração de Componentes”);
- `ccm_passivate`: utilizada para informar ao componente que este não se encontra ativo, podendo liberar recursos que tenha adquirido;
- `ccm_remove`: notifica o componente que este será destruído.

A Interface `SessionSynchronization`

É uma interface de *callback*, que pode ser implementada para que o componente seja notificado sobre os limites de transação. Sua sintaxe, em IDL3, é:

```
exception CCMException {CCMExceptionReason reason;};

local interface SessionSynchronization {
   void after_begin () raises (CCMException);
   void before_completion () raises (CCMException);
   void after_completion (in boolean committed) raises (CCMException);
};
```

A semântica das operações é:

- `after_begin`: informa que uma nova transação teve início;
- `before_completion`: informa que uma transação está para ser concluída;
- `after_completion`: informa que uma transação foi encerrada, com sucesso (*commit*) ou falha (*rollback*).

5.2.3.3 Interfaces para o Tipo de API de Contêiner Entidade

A Interface `EntityContext`

É uma interface interna, derivada de `CCMContext`, e fornece a uma instância do componente acesso aos serviços disponibilizados pelo contêiner. Sua sintaxe, em IDL3, é:

```
exception IllegalState { };

local interface EntityContext : CCMContext {
   Object get_CCM_object () raises (IllegalState);
   PrimaryKeyBase get_primary_key () raises (IllegalState);
};
```

A semântica das operações é:

- `get_CCM_object`: mesmo comportamento da operação de mesmo nome da interface `SessionContext`;
- `get_primary_key`: deve ser invocada no escopo de uma operação de *callback* (caso contrário, a exceção `IllegalState` será disparada), e devolve uma referência para a chave primária sendo utilizada pela instância do componente.

A Interface `EntityComponent`

É uma interface de *callback*, utilizada pelo contêiner como parte do gerenciamento do ciclo de vida dos serventes. Contém, ainda, operações para o tratamento de persistência de estado para componentes de processo ou entidade. Sua sintaxe, em IDL3, é:

```
exception CCMException {CCMExceptionReason reason;};

local interface EntityComponent : EnterpriseComponent {
   void set_entity_context (in EntityContext ctx) raises (CCMException);
   void unset_entity_context ()raises (CCMException);
   void ccm_activate () raises (CCMException);
   void ccm_load ()raises (CCMException);
   void ccm_store ()raises (CCMException);
   void ccm_passivate ()raises (CCMException);
   void ccm_remove ()raises (CCMException);
};
```

A semântica das operações é:

- `set_entity_context`: após criar uma instância do componente, mas fora do contexto de transações, o contêiner chama esta operação para informar ao componente o contexto em que este é executado;
- `unset_entity_context`: chamada para desassociar o contexto do componente. É invocada antes que a instância do componente seja destruída;

- `ccm_activate`: chamada pelo contêiner para informar ao componente que este foi ativado. A configuração/inicialização do componente deve ter sido realizada antes que esta operação seja invocada (seção 4.11, “Configuração de Componentes”);
- `ccm_load`: chamada pelo contêiner para que o componente carregue seu estado a partir do meio de persistência (como um banco de dados, por exemplo). Se o gerenciamento de persistência for realizado pelo contêiner (CMP), usando como mecanismo o Serviço de Persistência CORBA (PSS), esta operação pode ser gerada automaticamente. Caso o gerenciamento seja realizado pelo componente (SMP), a implementação desta operação é responsabilidade do desenvolvedor;
- `ccm_store`: chamada pelo contêiner para que o componente salve seu estado em meio persistente;
- `ccm_passivate`: utilizada para informar ao componente que este não se encontra ativo, podendo liberar recursos que tenha adquirido;
- `ccm_remove`: notifica o componente que este será destruído.

5.2.4 Interfaces de Programação para Componentes Servidores Estendidos

Assim como ocorre com componentes servidores básicos, para os estendidos há um conjunto de interfaces internas e de *callback*, para a comunicação entre o componente e o contêiner.

Diferentemente dos componentes básicos, os estendidos admitem portas e outras funcionalidades específicas dos Componentes CORBA, como segmentos e *proxies* (seção 6.1.6, “Construções para Otimização do Uso de Recursos”). Essas funcionalidades estão no escopo das interfaces aqui apresentadas.

5.2.4.1 Interfaces Comuns aos Tipos de APIs dos Contêineres

A Interface `CCM2Context`

É uma interface interna, derivada de `CCMContext` (seção 5.2.3.1, “Interfaces Comuns aos Tipos de APIs dos Contêineres”), que adiciona funções específicas de componentes estendidos, dando suporte ao Serviço de Persistência de Estado CORBA (PSS) e ao gerenciamento de referências e serventes usando o POA. Sua declaração, em IDL3, é:

```
typedef CosPersistentState::CatalogBase CatalogBase;
typedef CosPersistentState::TypeId TypeId;
```

```
exception PolicyMismatch { };
exception PersistenceNotAvailable { };

local interface CCM2Context : CCMContext {
   HomeRegistration get_home_registration ();
   void req_passivate () raises (PolicyMismatch);
   CatalogBase get_persistence (in TypeId catalog_type_id)
      raises (PersistenceNotAvailable);
};
```

A semântica das operações é:

- `get_home_registration`: obtém referência à interface `HomeRegistration` (a seguir, nesta seção);
- `req_passivate`: utilizada pelo componente para informar ao contêiner que o componente deseja ser desativado. Apenas para componentes com política de gerenciamento de tempo de vida dos serventes (seção 5.2.1.3, "Gerenciamento do Ciclo de Vida") "componente" ou "contêiner";
- `get_persistence`: devolve referência ao catálogo identificado pelo parâmetro `catalog_type_id`.

A Interface `HomeRegistration`

É uma interface interna utilizada por um componente CORBA para registrar seu *home*, tal que este possa ser encontrado através das operações da interface `HomeFinder` (seção 4.10, "*Home Finders*"). Sua declaração, em IDL3, é:

```
local interface HomeRegistration {
   void register_home (in CCMHome home_ref, in string home_name);
   void unregister_home (in CCMHome home_ref);
};
```

A semântica das operações é:

- `register_home`: registra a referência `home_ref` com o `HomeFinder`. Se o parâmetro `home_name` for diferente de `NULL`, este pode ser utilizado pela operação `HomeFinder::find_home_by_name`. Caso seja `NULL`, o *home* não poderá ser localizado por nome. A referência `home_ref` pode ser utilizada pelas operações:
  - `CCMHome::get_component_def` (seção 4.9.9, "A Interface `CCMHome`"): para obtenção de referência a `CORBA::ComponentIR::ComponentDef`. Esta, então, pode ser utilizada por `HomeFinder::find_home_by_component_type`;

  - `CORBA::Object::get_interface_def`: para obtenção de referência a `CORBA::InterfaceDef`, a ser utilizada por `HomeFinder::find_home_by_home_type`;
- `unregister_home`: remove o *home* do `HomeFinder`.

A Interface `ProxyHomeRegistration`

É uma interface interna, derivada de `HomeRegistration`, utilizada para registrar um *home* remoto, isto é, que não está colocado com as instâncias de componentes que este crie. Isto permite balanceamento de carga, aumentando a escalabilidade da solução (seção 6.1.6.2, "*Proxy Homes*"). Sua declaração, em IDL3, é:

```
exception UnknownActualHome { };
exception ProxyHomeNotSupported { };

local interface ProxyHomeRegistration : HomeRegistration {
   void register_proxy_home (in CCMHome rhome, in CCMHome ahome)
      raises (UnknownActualHome, ProxyHomeNotSupported);
};
```

A operação `register_proxy_home` registra o *proxy home* (`rhome`) com o `HomeFinder` e o mapeia ao *home* real (`ahome`), que já deve ter sido previamente registrado.

A Interface `Event`[8]

Conforme visto na seção 4.7, "Eventos", o contêiner é responsável pela interação entre os eventos do CCM e o Serviço de Notificação CORBA. A interface `Event` é um subconjunto deste último, fornecendo operações para subscrição e publicação de eventos, além do tratamento de canal de eventos com o Serviço de Notificação CORBA. Sua declaração, em IDL3, é:

```
typedef CosNotification::EventHeader EventHeader;
typedef CosNotifyChannnelAdmin::ChannelId Channel;
exception ChannelUnavailable { };
exception InvalidSubscription { };
exception InvalidName { };
exception InvalidChannel { };

local interface LocalCookie {
   boolean same_as (in LocalCookie cookie);
```

[8] Vale notar que a interface `Event` aqui presente difere da interface `Events` (seção 4.7.7, "A Interface `Events`").

```
};

local interface Event {
   EventConsumerBase create_channel (out Channel chid)
      raises (ChannelUnavailable);
   LocalCookie subscribe (in EventConsumerBase ecb,in Channel chid)
      raises (ChannelUnavailable);
   void unsubscribe (in LocalCookie cookie) raises (InvalidSubscription);
   EventConsumerBase obtain_channel (in string supp_name,in EventHeader hdr)
      raises (InvalidName);
   void listen (in EventConsumerBase ecb, in string csmr_name)
      raises (InvalidName);
   void push (in EventBase evt);
   void destroy_channel (in Channel chid) raises (InvalidChannel);
};
```

A semântica das operações é:

- `create_channel`: cria um canal com o Serviço de Notificação CORBA, identificado pelo parâmetro de saída `chid`;
- `subscribe`: subscreve um consumidor de eventos ao canal `chid`. Devolve um *cookie* que identifica a subscrição;
- `unsubscribe`: desfaz a subscrição, a partir do parâmetro `cookie`;
- `obtain_channel`: devolve referência ao canal de eventos;
- `listen`: aguarda a ocorrência de um evento;
- `push`: publica um evento;
- `destroy_channel`: remove o canal identificado por `chid`.

5.2.4.2 Interfaces para o Tipo de API de Contêiner Sessão

A Interface `Session2Context`

É uma interface interna, derivada de `SessionContext` (seção 5.2.3.2, "Interfaces para o Tipo de API de Contêiner Sessão"). Possui operações para tratamento de referências às interfaces de um componente CORBA. Sua declaração, em IDL3, é:

```
enum BadComponentReferenceReason {
   NON_LOCAL_REFERENCE,
   NON_COMPONENT_REFERENCE,
   WRONG_CONTAINER,
};
exception BadComponentReference {BadComponentReferenceReason reason;};
exception IllegalState { };

local interface Session2Context : SessionContext, CCM2Context {
   Object create_ref (in CORBA::RepositoryId repid);
   Object create_ref_from_oid (in CORBA::OctetSeq oid,
```

```
      in CORBA::RepositoryId repid);
   CORBA::OctetSeq get_oid_from_ref (in Object objref)
      raises (IllegalState, BadComponentReference);
};
```

A semântica das operações é:

- `create_ref`: devolve referência a ser exportada aos clientes, a partir de `repid`, que identifica o componente em si, uma das interfaces admitidas por herança, uma de suas bases ou uma de suas facetas;
- `create_ref_from_oid`: assim como em `create_ref`, devolve referência a ser exportada aos clientes. Entretanto, tal referência encapsula a informação extra contida em `oid`, que pode ser utilizada para invocar operações;
- `get_oid_from_ref`: obtém a informação extra armazenada na referência recebida como parâmetro.

5.2.4.3 Interfaces para o Tipo de API de Contêiner Entidade

A Interface `ComponentId`

É uma interface interna que encapsula informação de identidade, denominada identificador de componente, a ser trocada entre o contêiner e a implementação do componente.

O identificador de componente contém a seguinte informação:

- Identificador de faceta, que é uma referência à interface equivalente ou a uma das facetas do componente. Esta referência é denominada faceta alvo;
- Identificador de segmento (detalhes sobre segmentos na seção 6.1.6.1, “Executores Monolíticos e Segmentados”), que indica o segmento que contém a faceta alvo. Também chamado de segmento alvo;
- Uma seqüência de descritores de segmento, cada um contendo um identificador de segmento e um identificador de estado, que representa o estado de persistência do segmento em meio de armazenamento.

Quando a operação `ccm_load` for chamada pelo contêiner, sua implementação pode utilizar o identificador do componente para localizar e encarnar o estado do componente.

A interface `ComponentId` contém as seguintes operações:

- `get_target_facet`: devolve o identificador da faceta alvo;

- `get_target_segment`: devolve o identificador do segmento que contém a faceta alvo;
- `get_target_state_id`: devolve o identificador de estado do segmento alvo, a ser utilizado na interação com o mecanismo de persistência empregado;
- `get_segment_state_id`: dado um identificador de segmento, devolve o identificador de estado daquele segmento;
- `get_segment_descrs`: devolve uma seqüência de descritores de segmento contidos no identificador do componente;
- `create_with_new_target`: cria um novo identificador de componente, idêntico ao utilizado na chamada da operação, porém com faceta e segmento alvo novos. Esta operação pode ser útil na implementação de operações de navegação.

A Interface `Entity2Context`

É uma interface interna, derivada de `EntityContext` (seção 5.2.3.3, "Interfaces para o Tipo de API de Contêiner Entidade"), que acrescenta operações para tratamento de identificadores de componentes. Sua declaração, em IDL3, é:

```
exception BadComponentReference {BadComponentReferenceReason reason; };
exception IllegalState { };

local interface Entity2Context : EntityContext, CCM2Context {
   ComponentId get_component_id () raises (IllegalState);
   ComponentId create_component_id (
      in FacetId target_facet,
      in SegmentId target_segment,
      in SegmentDescrSeq seq_descrs);
   ComponentId create_monolithic_component_id (
      in FacetId target_facet, in StateIdValue sid);
   Object create_ref_from_cid (in CORBA::RepositoryId repid,in ComponentId cid);
   ComponentId get_cid_from_ref (in Object objref)raises(BadComponentReference);
};
```

A semântica das operações é:

- `get_component_id`: devolve uma referência ao identificador do componente;
- `create_component_id`: cria um identificador de componente. Aqui, `target_facet` identifica a faceta alvo, `target_segment` identifica o segmento que contém a faceta alvo e `seq_descrs` contém os descritores de segmentos;

- `create_monolithic_component_id`: é uma forma simplificada da operação `create_component_id`, utilizada na criação de identificadores de componentes para executores monolíticos (seção 6.1.6.1, “Executores Monolíticos e Segmentados”);
- `create_ref_from_cid`: a partir de um identificador de componente (obtido com a operação `get_component_id`), cria uma referência a objeto que o encapsule para ser exportada aos clientes;
- `get_cid_from_ref`: devolve o identificador de componente encapsulado na referência a objeto recebida como parâmetro.

5.3 Considerações sobre a Programação de Clientes

O foco desta seção é apresentar como um cliente CORBA, que não seja um componente, pode interagir com componentes CORBA, que são executados em servidores de componentes.

Um cliente pode interagir inicialmente com um componente CORBA através de dois tipos de interfaces externas: `Home` e interfaces de aplicação. Além disso, um cliente pode ser classificado como:

- Ciente de componente, ou seja, um cliente que sabe que está interagindo com um componente CORBA e, portanto, pode utilizar todas as funcionalidades por aquele disponibilizadas;
- Não ciente de componente, isto é, um cliente que não sabe que está interagindo com um componente CORBA. Deste modo, pode apenas utilizar funcionalidades disponibilizadas por objetos CORBA, não tendo acesso, portanto, a funcionalidades específicas de componentes CORBA.

Uma conseqüência desta classificação é: clientes não cientes de componentes podem utilizar apenas componentes básicos, e clientes cientes de componentes podem utilizar componentes básicos e estendidos.

Os clientes podem utilizar dois padrões para acessar um componente CORBA: *factory* e *finder*, este último apenas se o componente possuir chave primária. Esta seção apresenta a utilização destes padrões, bem como o uso de transações e segurança.

5.3.1 Clientes Cientes de Componentes (*Component-aware*)

Esses clientes são definidos usando extensões em IDL3, e podem interagir com componentes CORBA através das seguintes interfaces:

- Interface equivalente;
- Interfaces admitidas por herança;

- Facetas;
- Interface `Home`.

Referências a essas interfaces podem ser obtidas a partir de uma referência ao `HomeFinder` (seção 4.10, "*Home Finders*") ou ao serviço de nomes. Essas e outras referências utilizadas por um cliente para interagir com um componente CORBA, são obtidas através da operação `resolve_initial_references`, em `CORBA::ORB`. Essa operação admite as seguintes referências (em parênteses estão os valores que a operação deve receber como parâmetro), sendo o ponto de partida para as interações entre o cliente e o componente:

- Serviço de nomes ("`NameService`");
- HomeFinder ("`ComponentHomeFinder`");
- Repositório de interfaces ("`InterfaceRepository`");
- Contexto de transação ("`TransactionCurrent`"): devolve uma referência a partir da qual pode-se utilizar operações transacionais, como confirmação ou cancelamento de transações;
- Contexto de segurança ("`SecurityCurrent`"): devolve uma referência a ser utilizada para restringir acesso a funcionalidades do componente;
- Serviço de notificação ("`NotificationService`").

5.3.1.1 Utilização de *Factory*

A partir do `HomeFinder`, o cliente deve obter referência ao *home* do componente para, então, utilizar a operação `create`, definida na interface implícita do componente, ou outra operação, definida pelo desenvolvedor do componente na interface explícita (seção 4.9, "*Homes*"). O trecho de código a seguir ilustra o uso da operação `create`:

```
// Obtém referência ao HomeFinder
org.omg.CORBA.Object objhf =
   orb.resolve_initial_references("ComponentHomeFinder");
ComponentHomeFinder hf = ComponentHomeFinderHelper.narrow(objhf);

// Localiza o home do componente A
org.omg.CORBA.Object objh = hf.find_home_by_type(AHomeHelper.id());
AHome h = AHomeHelper.narrow(objh);

// Cria uma instância do componente
org.omg.Components.ComponentBase abase = h.create();
A meua = AHelper.narrow(abase);

// Utiliza operações do componente
res = meua.foo(param);
```

5.3.1.2 Utilização de *Finder*

Para que um cliente obtenha referência a uma instância pré-existente de um componente CORBA (apenas da categoria entidade), deve utilizar operações de *finder*, através das operações da interface explícita do *home* do componente, obtido a partir do `HomeFinder`, como no exemplo da seção 5.3.1.1, ”Utilização de *Factory*”, substituindo-se a operação `create` por uma operação de *finder*. Outra possibilidade é a utilização do serviço de nomes para a localização de uma instância do componente previamente lá registrada, conforme o exemplo a seguir:

```
// Obtém referência ao serviço de nomes
org.omg.CORBA.Object objns =
   orb.resolve_initial_references(“NamingService”);
NamingContext nc = NamingContextHelper.narrow(objns);

// A é o nome do componente. Obtém referência a instância de A
// previamente registrada
NameComponent ncomp = new NameComponent(“A“,””);
NameComponent path[] = {ncomp};
A meua = AHelper.narrow(nc.resolve(path));

// Utiliza operações do componente
res = meua.foo(param);
```

Detalhes sobre a utilização do serviço de nomes em Java podem ser obtidos em [BVD01, seção 7.1, “*The CORBA Naming Service*”].

5.3.1.3 Utilização de Transações

Um cliente pode invocar operações de um componente no contexto de transações. Para isso, deve obter referência ao contexto de transação corrente e, então, utilizar as operações específicas da interface `UserTransaction` (seção 5.2.3.1, “Interfaces Comuns aos Tipos de APIs dos Contêineres”). O trecho de código a seguir ilustra tal utilização:

```
// Obtém o contexto de transação
org.omg.CORBA.Object objref =
   orb.resolve_initial_references(“TransactionCurrent”);
Current txRef = CurrentHelper.narrow(objRef);

// Inicia nova transação
txRef.begin();

// Utiliza operações do componente
res = meua.foo(param);

// Encerra a transação
txRef.commit();
```

5.3.1.4 Utilização de Segurança

A utilização de mecanismos de segurança CORBA a partir do cliente deve ser feita através de SSL ou SECIOP [SS02]. O trecho de código abaixo ilustra a utilização deste último:

```
// Obtém referência ao contexto de segurança corrente.
org.omg.CORBA.Object objsc =
   orb.resolve_initial_references("SecurityCurrent");
org.omg.SecurityLevel2.PrincipalAuthenticator pa =
   org.omg.SecurityLevel2.PrincipalAuthenticatorHelper.narrow(objsc);

// Autentica o usuário
pa.authenticate(...);

// Utiliza operações do componente
res = meua.foo(param);
```

5.3.1.5 Utilização de Eventos

Clientes CORBA podem utilizar eventos conforme a seção 4.7, "Eventos", ou podem ainda utilizar o Serviço de Notificação CORBA [NSS02] diretamente, conforme o exemplo a seguir:

```
// Obtém referência ao serviço de notificação
org.omg.CORBA.Object objns =
   orb.resolve_initial_references("NotificationService");
org.omg.CosNotififyChannelAdmin.EventChannelFactory ecf =
   org.omg.EventChannelFactoryHelper.narrow(objns);

// Cria um canal de eventos
org.omg.CosNotifyChannelAdmin.EventChannel ec =
   ecf.create_channel(...);

// Obtém um SupplierAdmin e um ConsumerProxy
org.omg.CosNotifyChannelAdmin.SupplierAdmin publisher =
   ec.new_for_suppliers(...);
org.omg.CosNotifyComm.ProxyConsumer proxy =
   publisher.obtain_notification_push_comsumer (...);

// Publica um evento
proxy.push_structured_event(...);
```

5.3.2 Clientes Não Cientes de Componentes (*Component-unaware*)

Os clientes que não sabem que interagem com componentes são, a rigor, objetos CORBA comuns. Assim, o início de sua interação com os componentes provavelmente não será através do uso de `resolve_initial_references("ComponentHomeFinder")`, embora nada

o proíba. O que deve ocorrer é o uso dos mesmos mecanismos, tanto para *factory* quanto para *finder*, utilizados para acesso a objetos CORBA.

5.3.2.1 Utilização de *Factory*

Apenas se operações de *factory* forem definidas nas interfaces admitidas por herança. Neste caso, existem vários modos através dos quais um cliente pode acessar um componente:

- Referência a um *factory finder* previamente registrado com o serviço de nomes ou *trading* [TRAD00]. Em geral, `CosLifeCycle::FactoryFinder`;
- Referência ao *home* previamente registrado com o serviço de nomes ou *trading*;
- Referência convertida para `string` e armazenada em arquivo.

5.3.2.2 Utilização de *Finder*

Através do serviço de nomes, conforme ilustrado na seção 5.3.1.2, “Utilização de *Finder*”.

5.3.2.3 Utilização de Transações e Segurança

O mesmo dos clientes cientes de componentes, descritos nas seções 5.3.1.3, “Utilização de Transações” e 5.3.1.4, “Utilização de Segurança”.

5.3.2.4 Utilização de Eventos

Através do serviço de notificação, conforme ilustrado na seção 5.3.1.5, “Utilização de Eventos”.

Capítulo 6

CCM Implementation Framework – CIF

CCM define grande número de interfaces para comportar a estrutura e a funcionalidade dos componentes como, por exemplo:

- Interface `Home`, com as operações de *finder* e *factory*;
- Interface `Navigation`, que permite a obtenção de referência às facetas ou à interface equivalente, através de uma referência válida a alguma interface do componente;
- Interfaces `Receptacles` e `Events`.

As implementações de muitas dessas interfaces podem ser geradas automaticamente: esse é o objetivo do Arcabouço para Implementação de Componentes CORBA (*CORBA Component Implementation Framework, CIF*).

CCM define uma linguagem declarativa, *Component Implementation Definition Language (CIDL)*, para descrever implementações e persistência de estado de componentes e *homes*.

CIF usa a CIDL para gerar esqueletos que automatizam tarefas básicas, como navegação, ativação e gerenciamento de estado.

A figura abaixo representa a seqüência de desenvolvimento/compilação de componentes CCM, através do CIF.

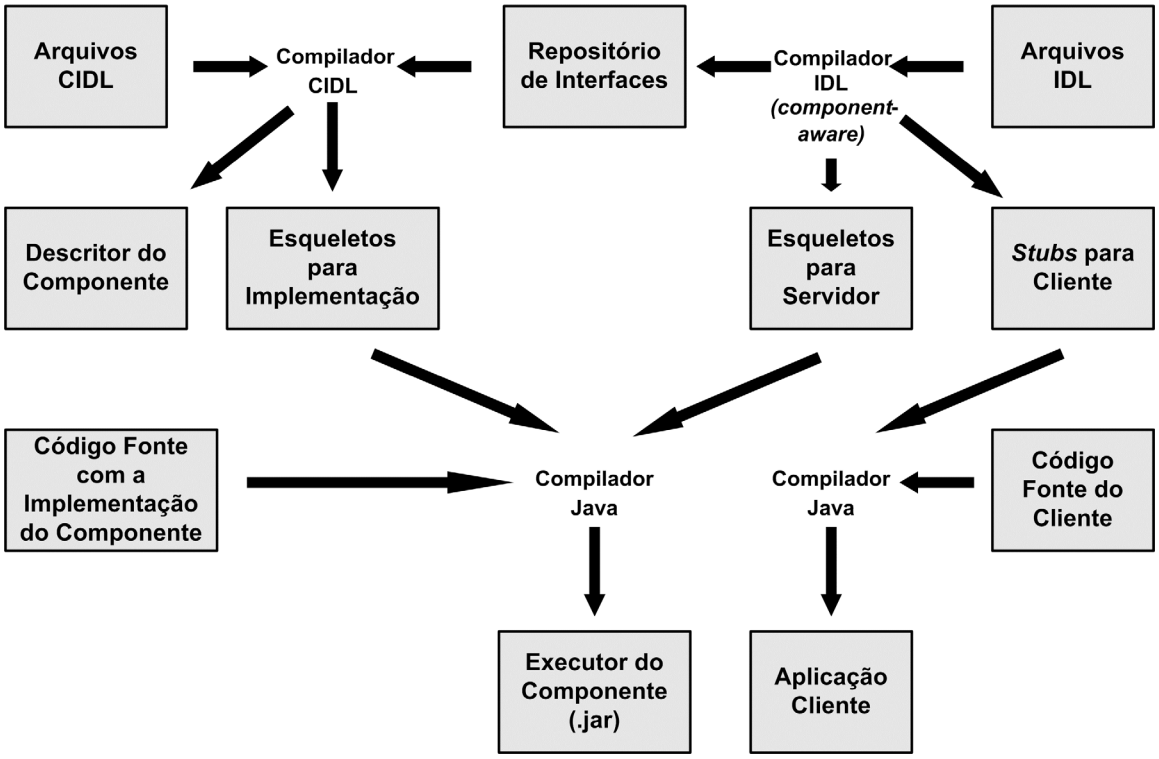

Figura 6.1: *CCM Implementation Framework (CIF)*

Este capítulo descreve as principais construções de um arquivo CIDL, com a apresentação de conceitos ilustrados através de exemplos.

6.1 Composições

A implementação de um componente envolve um conjunto de artefatos[9] que deve apresentar relacionamentos e comportamentos no sentido de constituir uma unidade que seja funcional e completa. Esse conjunto envolve dois tipos básicos de artefatos: os que são gerados automaticamente e os que são implementados pelo programador.

Os artefatos gerados automaticamente fornecem a infra-estrutura básica de um componente, como esqueletos que automatizam navegação, consultas à identificação, ativação, gerenciamento de estado e de ciclo de vida do componente, e também a solução de questões como escalabilidade da solução, através de conceitos como *proxies* e segmentação (a serem vistos neste capítulo).

[9] No contexto de um componente CORBA implementado em uma linguagem orientada a objetos, o termo artefato ou artefato de programação equivale ao conceito de classe. Da mesma forma, o termo esqueleto, quando empregado no contexto do CIF, equivale a uma classe abstrata a ser estendida para completar a implementação de um componente ou *home*. Esses termos objetivam uma forma genérica que seja válida também para linguagens procedurais.

Já os artefatos a serem implementados pelo programador constituem o comportamento do componente, inerente ao negócio. Estes representam geralmente uma pequena parte do número total de artefatos.

Ao conjunto de artefatos e sua definição denomina-se composição.

6.1.1 Executores

O termo executor é empregado para identificar os artefatos de programação responsáveis pela implementação do comportamento do componente ou de seu *home*.

6.1.1.1 Executores de Componentes

Quando uma definição de composição, em CIDL, é compilada, um esqueleto de implementação do componente é produzido, contendo implementações das operações oriundas de interfaces que compõem a estrutura do CCM, como `CCMObject`, por exemplo. Esse esqueleto também possui declarações de operações de negócio, vindas da declaração do componente em IDL. Essas operações devem ser implementadas pelo programador.

Em Java, por exemplo, o esqueleto gerado é uma classe abstrata, que deve ser herdada e ter as operações de negócio implementadas.

6.1.1.2 Executores de *Homes*

Diferentemente do que ocorre com os executores de componentes, a compilação de uma composição gera a implementação do *home* do componente, ao invés de um esqueleto.

Em Java, o artefato gerado é uma classe que pode ser estendida pelo programador.

6.1.2 Estrutura Básica de Composições

Composições são representadas em CIDL pelo meta-tipo `composition`. A definição básica de uma composição, em CIDL, é:

```
composition <category> <composition_name> {
   home executor <home_executor_name> {
      implements <home_type>;
      manages <executor_name>;
   };
};
```

onde:

- `<category>` indica a categoria do componente (seção 5.1.4, "Categorias de Componentes") a que se refere a composição, identificando o ciclo de vida daquele. Os possíveis valores são `entity`, `process`, `service` e `session`;
- O tipo de *home* do componente, identificado por `<home_type>`, implicitamente define o tipo de componente gerenciado, uma vez que um tipo de *home* gerencia apenas um tipo de componente (seção 4.9, "*Homes*"). Assim, torna-se desnecessário explicitar o tipo de componente na declaração da composição;
- Os executores do componente e do *home*, identificados por `<executor_name>` e `<home_executor_name>`, nesta ordem, serão gerados durante a compilação da declaração acima.

6.1.2.1 Exemplo

Este exemplo ilustra a utilização dos conceitos básicos de composições, e apresenta também os arquivos necessários para o desenvolvimento de uma aplicação simples, a partir das declarações em IDL e CIDL. Estão representados os artefatos gerados automaticamente e os que são de responsabilidade do programador. Vários trechos de código foram omitidos, como implementação de tratamento de exceções e implementações geradas em operações implícitas, no intuito de facilitar a compreensão do exemplo.

Os artefatos apresentados neste capítulo apenas ilustram o resultado da compilação por uma implementação hipotética do CCM, sendo que implementações reais podem gerar artefatos diferentes. De fato, a implementação EJCCM, utilizada para a construção do exemplo citado na seção 3.1, "Exemplo de Utilização", produz um conjunto distinto do apresentado neste capítulo. Optamos por não apresentar aqui os artefatos gerados pela implementação EJCCM devido à sua maior complexidade, que dificulta a compreensão, fugindo ao objetivo deste texto.

Definição do componente, em IDL

```
// IDL - PROGRAMADOR
//
module Alpinismo {
   interface Alpinista {
      void escala (in long distancia);
   };
   component Escalada {
      // Homenagem ao alpinista brasileiro Waldemar Niclevicz
      provides Alpinista niclevicz;
   };
   // CBA: Confederação Brasileira de Alpinismo
   home CBA manages Escalada {};
};
```

A compilação desta definição resulta no seguinte código, em Java:

```
// Interfaces geradas a partir da compilação das definições em IDL
//
package Alpinismo;
import org.omg.Components.*;
public interface AlpinistaOperations {
   public void escala (int distancia);
}
public interface EscaladaOperations
   extends CCMObjectOperations {
      Alpinismo.Alpinista provide_niclevicz();
}
public interface CBAExplicitOperations
   extends CCMHomeOperations { }
public interface CBAImplicitOperations
   extends KeylessCCMHomeOperations {
      Escalada create();
}
public interface CBAOperations extends
   CBAExplicitOperations, CBAImplicitOperations {}
```

Definição da composição em CIDL

```
// CIDL - PROGRAMADOR
//
import ::Alpinismo;
module Esportes {
   composition session EscaladaImpl {
      home executor CBAImpl {
         // o nome do home abaixo implicitamente define
         // o componente a ser implementado
         implements Alpinismo::CBA;
         manages EscaladaSessionImpl;
      };
   };
};
```

Neste exemplo, `EscaladaImpl` é o nome da composição, dando suporte a componentes de sessão (para maiores detalhes, vide 5.1.4, “Categorias de Componentes”). O nome do executor de *home* a ser gerado é `CBAImpl`, e este irá conter a implementação do *home* `CBA`, importado da definição em IDL. A composição ainda define o nome do executor do componente a ser gerado, `EscaladaSessionImpl`, a ser gerenciado pelo executor do *home*.

A compilação das declarações CIDL acima resulta nos seguintes artefatos:

- O esqueleto do executor do componente, `EscaladaSessionImpl`;
- A implementação do executor do *home*, `CBAImpl`.

Esses artefatos possuem, em Java, a seguinte forma:

Esqueleto do executor do componente

```
// Esqueleto do executor do componente,
// gerado a partir das definições em CIDL
//
package Esportes;
import Alpinismo;
import org.omg.Components.*;

abstract public class EscaladaSessionImpl
   implements EscaladaOperations, SessionComponent, ExecutorSegmentBase
{
   // Foram omitidas as implementações de operações herdadas de
   // SessionComponent e ExecutorSegmentBase
   protected EscaladaSessionImpl() {
      // Implementação gerada pela compilação ...
   }
   // A operação abaixo deve ser implementada pelo programador.
   abstract public AlpinistaOperations _get_facet_niclevicz();
}
```

Implementação do executor do *home*

```
// Implementação do executor do home,
// gerado a partir das definições em CIDL
//
package Esportes;
import Alpinismo;
import org.omg.Components.*;

public class CBAImpl
   implements Alpinismo::CBAOperations, HomeExecutorBase, CCMHome {

   // Foram omitidas as implementações de operações herdadas de
   // HomeExecutorBase e CCMHome
   //
   // Implementações para as operações de Alpinismo::CBAOperations
   CCMObject create_component() {
      return create();
   }
   void remove_component(CCMObject comp) {}
   Escalada create() {}
   ...
}
```

Além dos artefatos acima, gerados automaticamente, existem ainda aqueles que devem ser implementados pelo programador, conforme o exemplo abaixo:

Implementação do executor do componente, a partir do esqueleto gerado

```
// Implementação realizada pelo programador do componente
//
import Alpinismo.*;
```

```
import Esportes.*;

public class minhaEscaladaImpl extends EscaladaSessionImpl
   implements AlpinistaOperations {

   protected long totalPercorrido;
   public minhaEscaladaImpl() {
      super();
      totalPercorrido = 0;
   }
   public void escala(int distancia) {
      totalPercorrido += distancia;
   }
   public AlpinistaOperations _get_facet_niclevicz() {
      return (AlpinistaOperations) this;
   }
}
```

Implementação do executor do *home*, a partir do artefato gerado

Este executor age como um *factory* do executor do componente, devendo implementar o método `create_executor_segment` (mais informações sobre segmentos na seção 6.1.6.1, “Executores Monolíticos e Segmentados”). O ponto de entrada da composição é o método `create_home_executor`, que também deve ser implementado pelo programador.

```
// Implementação realizada pelo programador do componente
//
import Alpinismo.*;
import Esportes.*;
public class minhaCBAImpl extends CBAImpl {
   protected minhaCBAImpl() {
      super();
   }
   // Este método age como um factory para o executor do componente
   ExecutorSegmentBase create_executor_segment (int segid) {
      return new minhaEscaladaImpl();
   }
   // Este método cria nova instância do executor de home
   public static ExecutorSegmentBase create_home_executor() {
      return new minhaCBAImpl();
   }
}
```

6.1.3 Composições com Armazenagem Gerenciada

Composições que implementam componentes classificados como entidade ou processo necessitam, como parte do gerenciamento de estado, tratar a persistência deste em meio de armazenamento não volátil.

6.1.3.1 Serviço de Persistência de Estado

O Serviço de Persistência de Estado [PSS01] define dois métodos para prover o armazenamento de estado de objetos:

- Através da *Persistent State Definition Language* (PSDL);
- Através de Persistência Transparente, onde a implementação do objeto gerencia seu estado diretamente.

A linguagem PSDL é um superconjunto da OMG IDL, com quatro novas construções:

- `abstract storagetype`: define um tipo para armazenamento de estado. Em Java, é representado por uma interface;
- `storagetype`: implementa o tipo abstrato, sendo representado por uma classe em Java;
- `abstract storagehome`: define operações para gerenciamento do armazenamento. É representado em Java como uma interface;
- `storagehome`: implementa as operações definidas para gerenciamento, sendo representado por uma classe em Java.

A Figura 6.2 apresenta a relação entre as linguagens IDL, PSDL e CIDL.

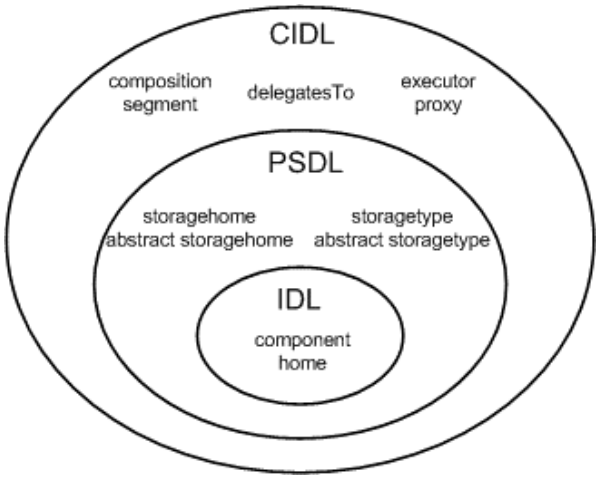

Figura 6.2: relação entre as linguagens IDL, PSDL e CIDL.
Estão ilustradas algumas palavras-chave específicas de cada linguagem.

O exemplo a seguir ilustra a definição em PSDL do armazenamento de estado de um objeto que representa uma pessoa:

```
// arquivo Pessoa.psdl
abstract storagetype Pessoa {
   readonly state long cpf;
```

```
   state string nome;
   state string endereco;
};
abstract storagehome PessoaHome of Pessoa {
   Pessoa create(in long cpf, in string nome, in string endereco);
};


// arquivo PessoaImpl.psdl
storagetype PessoaImpl implements Pessoa {};
storagehome PessoaHomeImpl of PessoaImpl implements PessoaHome{};
```

A ferramenta fornecida pela implementação do PSS deve gerar arquivos contendo interfaces e implementações (declarações, semelhante ao que ocorre com as composições CORBA, exemplificadas na seção 6.1.2.1, “Exemplo”) na linguagem desejada. A partir desses arquivos, o programador deve fornecer a codificação necessária, estendendo as implementações geradas.

Com Persistência Transparente, o mesmo exemplo deveria ser implementado diretamente pelo programador:

```
// arquivo Pessoa.java
// Definição do tipo de armazenamento
public interface Pessoa {
   public long cpf();
   public String nome();
   public void nome(String _nome);
   public String endereco();
   public void endereco(String _endereco);
}


// arquivo PessoaImpl.java
// Implementação do tipo de armazenamento
public class PessoaImpl implements Pessoa {
   private long _cpf;
   private String _nome;
   private String _endereco;
   public long cpf() {
      return _cpf ;
   }
   ...
}
```

Com Persistência Transparente, todavia, não é possível definir armazenamento de *homes* (`storagehome`). São implicitamente definidos armazenamentos padrão de *homes*, derivados de `java.lang.Object`, sem chaves ou operações.

6.1.3.2 Gerenciamento de Estado de Componentes

De maneira análoga ao que ocorre com os objetos CORBA, o estado dos componentes CORBA pode ser gerenciado diretamente por sua implementação.

Contudo, a presença de construções para gerenciamento de estado na declaração das composições fornece maior organização deste processo. A declaração, em CIDL, de uma composição deste tipo possui a forma[10]:

```
composition <category> <composition_name> {
   home executor <home_executor_name> {
      implements < home_type>;
      bindsTo <abstract_storage_home>;
      manages <executor_name>;
   };
};
```

A CIDL é um superconjunto da PSDL, contendo a construção `bindsTo`, necessária para prover o armazenamento de estado. Essa construção indica a associação do *home* do componente e, implicitamente, do componente em si, a um tipo abstrato de armazenamento, `<abstract_storage_home>`.

6.1.3.3 *Homes* sem Chave Primária

Para *homes* que não se utilizem de chaves primárias, o armazenamento de *homes* pode definir operações de *factory*. Por exemplo, a definição, em CIDL:

```
abstract storagetype Conta {
   state string numeroConta;
   state float saldo;
};
abstract storagehome Banco {
   factory create(numeroConta);
};
```

resulta na seguinte operação, em Java:

```
Conta create(in string numeroConta);
```

6.1.3.4 *Homes* com Chave Primária

Os *homes* com chave primária, além de possuírem as operações de *factory*, também possuem operações de *finder*, sendo que a definição, em CIDL:

```
abstract storagetype Pessoa {
   readonly state long cpf;
```

[10] A especificação atual do CCM [CCM02] não foi atualizada de acordo com a especificação da PSDL [PSS01], que eliminou a palavra-chave "`catalog`". Entretanto, este texto considera esta futura atualização, já discutida na lista de pendências disponível em http://www.omg.org/issues/components-ftf.html#Issue5499.

```
   state string nome;
   state string endereco;
};
abstract storagehome PessoaHome of Pessoa {
   key cpf;
};
```

resulta nas operações a seguir, em Java:

```
Pessoa find_by_cpf(int cpf);
PessoaRef find_ref_by_cpf(int cpf);
```

Para composições com armazenagem gerenciada, a implementação das operações de *finder* do *home* do componente é gerada automaticamente, em termos das operações acima. Essa funcionalidade é denominada “Delegação Implícita de Operações do *Home*”. Assim, no exemplo acima, a operação `find_by_primary_key` pode delegar a busca para `find_ref_by_cpf`. O mesmo se aplica às operações de *factory*. O exemplo da próxima seção ilustra tal delegação.

6.1.3.5 Exemplo

Este exemplo ilustra a utilização dos conceitos de persistência e chave primária, em um componente classificado como entidade (maiores detalhes sobre categorias de componentes na seção 5.1.4). O código a seguir estende o exemplo anterior (seção 6.1.2.1), sendo que os trechos comuns foram omitidos.

Definição do componente, em IDL

```
// IDL - PROGRAMADOR
//
module Alpinismo {
   valuetype CodigoEscalada: Components::PrimaryKeyBase {
      public string codigo;
   };
   home CBA manages Escalada primarykey CodigoEscalada{};
};
```

A compilação desta definição resulta no mesmo código do exemplo anterior, em Java, exceto por:

```
// Interfaces geradas a partir da compilação das definições em IDL
//
public interface CBAImplicitOperations {
   Escalada create(Alpinismo.CodigoEscalada key)
      throws DuplicateKey, InvalidKey;
   Escalada find_by_primary_key(Alpinismo.CodigoEscalada key)
      throws UnknownKey, InvalidKey;
```

```
   void remove(Alpinismo.CodigoEscalada key)
      throws UnknownKey, InvalidKey;
   Alpinismo.CodigoEscalada get_primary_key(Escalada comp);
}
```

Definição da composição em CIDL

```
// CIDL - PROGRAMADOR
//
import ::Alpinismo;
module Esportes {
   abstract storagetype EstadoEscalada {
      state Alpinismo::CodigoEscalada codigo;
      state string descricao;
   };
   abstract storagehome EstadoEscaladaHome of EstadoEscalada {
      key codigo;
      factory create(codigo);
   };
   composition entity EscaladaImpl {
      home executor CBAImpl {
         implements Alpinismo::CBA;
         bindsTo EstadoEscaladaHome;
         manages EscaladaEntityImpl;
      };
   };
};
```

Como no exemplo anterior, neste exemplo `EscaladaImpl` é o nome da composição. O nome do executor de *home* a ser gerado é `CBAImpl`, e este irá conter a implementação do *home* `CBA`, importado da definição em IDL. A composição ainda define o nome do executor do componente a ser gerado, `EscaladaEntityImpl`, e que será gerenciado pelo executor do *home*.

A compilação das declarações CIDL acima resulta nos seguintes artefatos:

- O esqueleto do executor do componente, `EscaladaEntityImpl`;
- A implementação do executor do *home*, `CBAImpl`;
- A interface para o tipo abstrato de armazenamento, `EstadoEscalada`;
- A interface para o *home* do armazenamento abstrato, `EstadoEscaladaHome`.

Esses artefatos possuem, em Java, a seguinte forma:

Esqueleto do executor do componente

```
// Esqueleto do executor do componente,
// gerado a partir das definições em CIDL
//
package Esportes;
```

```
import Alpinismo;
import org.omg.Components.*;

abstract public class EscaladaEntityImpl
       implements EscaladaOperations, PersistentComponent, ExecutorSegmentBase {

   // Foram omitidas as implementações de operações herdadas de
   // PersistentComponent e ExecutorSegmentBase
   protected EstadoEscaladaIncarnation _state;

   protected EscaladaEntityImpl() {
      _state = null;
   }

   public void set_incarnation(EstadoEscalada state) {
      _state = state;
   }
   // A operação abaixo deve ser implementada pelo programador.
   abstract public AlpinistaOperations _get_facet_niclevicz();
}
```

Implementação do executor do *home*

```
// Implementação do executor do home,
// gerado a partir das definições em CIDL
//
package Esportes;
import Alpinismo;
import org.omg.Components.*;

public class CBAImpl
   implements Alpinismo::CBAOperations, PersistentComponent,
      ExecutorSegmentBase {

   // os valores para as variáveis a seguir devem ser atribuídos
   // durante a inicialização e ativação
   protected Entity2Context _origin;
   protected EstadoEscaladaHome _storageHome;
   ...

   // Foram omitidas as implementações de operações herdadas de
   // PersistentComponent e ExecutorSegmentBase
   //
   // Implementações para as operações de Alpinismo::CBAOperations
   Escalada create(Alpinismo.CodigoEscalada key)
      throws DuplicateKey, InvalidKey {

      // delegação da criação do home de armazenamento para a operação
      // create de EstadoEscaladaHome
      EstadoEscalada new_state = _storageHome.create(key);
      ...
   }

   Escalada find_by_primary_key(Alpinismo.CodigoEscalada key)
      throws UnknownKey, InvalidKey {

      // delegação da busca para a operação
      // find_ref_by_codigo de EstadoEscaladaHome
      EstadoEscaladaRef ref = _storageHome.find_ref_by_codigo(key);
      ...
   }
```

```
   void remove(Alpinismo.CodigoEscalada key)
      throws UnknownKey, InvalidKey {
      ...
   }

   Alpinismo.CodigoEscalada get_primary_key(Escalada comp) {
      ...
   }
   ...
}
```

Interface para o tipo abstrato de armazenamento

```
package Esportes;
import org.omg.CosPersistentState.*;
import Alpinismo.*;
public interface EstadoEscalada extends StorageObject {
   public string descricao();
   public void descricao(String val);
}
```

Interface para o *home* do armazenamento abstrato

```
public interface EstadoEscaladaHome extends StorageHomeBase {
   public EstadoEscalada find_by_codigo(CodigoEscalada k);
   public EstadoEscaladaRef find_ref_by_codigo(CodigoEscalada k);
}
```

Nos artefatos a serem implementados pelo programador, pode-se utilizar informações contidas no estado do componente, como destacadas em negrito no exemplo:

Implementação do executor do componente, a partir do esqueleto gerado

```
// Implementação realizada pelo programador do componente
//
import Alpinismo.*;
import Esportes.*;

public class minhaEscaladaImpl extends EscaladaEntityImpl
   implements AlpinistaOperations {

   protected long totalPercorrido;
   public minhaEscaladaImpl() {
      super();
   totalPercorrido = 0;
   }
   public void escala(long distancia) {
      totalPercorrido += distancia;
      System.out.println("Total percorrido na escalada " + _state.descricao() +
         ": " + totalPercorrido);
```

```
   }
   public AlpinistaOperations _get_facet_niclevicz() {
      return (AlpinistaOperations) this;
   }
}
```

6.1.4 Delegação Explícita de Operações

Conforme descrito na seção 4.9.8, “Operações Ortodoxas e Heterodoxas”, a geração de implementação para as operações ortodoxas é automática. Não obstante, o desenvolvedor pode optar por codificar uma operação ortodoxa suprimindo sua geração através da construção `abstract` na declaração do executor do *home*. Neste caso, será gerada apenas a declaração da operação como abstrata, ficando a codificação a cargo do desenvolvedor.

Segue a sintaxe da construção `abstract`, destacada em negrito:

```
composition <category> <composition_name> {
   home executor <home_executor_name> {
      implements < home_type>;
      bindsTo <abstract_storage_home>;
      manages <executor_name>;
      abstract(<home_op1>, <home_op2>, …);
   };
};
```

Quanto às operações heterodoxas, a menos do caso citado na seção 6.1.3.4, “*Homes* com Chave Primária”, que aborda delegação implícita, as implementações não são automaticamente geradas, ficando tal tarefa a cargo do programador. Todavia, a CIDL possui a construção `delegatesTo`, para permitir que seja especificado como operações heterodoxas serão implementadas. Existem duas formas para essa construção, destacadas em negrito na declaração:

```
composition <category> <composition_name> {
   home executor <home_executor_name> {
      implements < home_type>;
      bindsTo <abstract_storage_home>;
      manages <executor_name>;
      delegatesTo abstract storagehome (
         <home_op0> : <storage_home_op0>,
         <home_op1> : <storage_home_op1>, ...
      );
      delegatesTo executor(
         <home_op2> : <executor_op2>, ...
      );
   };
};
```

- `delegatesTo abstract storagehome`: essa construção define um mapeamento entre operações no executor do *home*,

representadas na declaração acima por `<home_op0>` e `<home_op1>`, e operações no armazenamento do *home*, `<storage_home_op0>` e `<storage_home_op1>`. A partir dessa declaração, o CIF gera implementações, no executor do *home*, que delegam para operações no armazenamento do *home*;

- `delegatesTo executor`: define um mapeamento entre operações do *home* e do executor do componente. A operação do *home* deve ser de *factory*, como uma operação explícita, ou ainda a operação `create`, representando uma operação de *factory* implícita. Essa obrigatoriedade das operações serem de *factory* deve-se ao fato de que outras operações de *home* não possuem componente alvo, ou seja, não se referem a um componente específico. Por ser uma operação de *factory*, sua implementação, gerada automaticamente, criará uma instância do componente para, então, invocar a operação no executor deste. Tal operação é disponibilizada apenas ao *home* do componente, em uma interface chamada faceta de *factory*. A denominação faceta é empregada porque através dessa interface o componente disponibiliza funcionalidades. Entretanto, tal interface não é propriamente uma faceta, como definido na seção 4.4, "Facetas (*Facets* ou *Provided Interfaces*)", sendo que é exposta apenas ao *home*, não sendo visível pelos clientes, nem mesmo através da operação `Navigation::get_all_facets`. A principal razão para este tipo de delegação é permitir a inicialização de estado, a ser compartilhado pelas instâncias do componente.

6.1.5 Delegação de Portas e Atributos

Quando um componente possui estado, é possível delegar operações em portas como receptáculos, atributos e fontes de eventos para membros do gerenciador abstrato de *homes*. Tal funcionalidade é definida, em CIDL, pela construção `delegatesTo abstract storagetype`, destacada em negrito na declaração:

```
composition <category> <composition_name> {
   home executor <home_executor_name> {
      implements < home_type>;
      bindsTo <abstract_storage_home>;
      manages <executor_name>;
      delegatesTo abstract storagetype (
         <feature_name0> : <storage_member_name0>,
         <feature_name1> : <storage_member_name1>, ...
      );
   };
};
```

A partir dessa declaração, o CIF gera implementações específicas para cada tipo de porta, como operações para conexão e desconexão a receptáculos.

6.1.6 Construções para Otimização do Uso de Recursos

As seções anteriores descrevem funcionalidades que conferem comportamentos específicos para a utilização dos componentes CORBA. O restante deste capítulo apresenta construções, na definição de composições, que aumentam a flexibilidade de soluções componentizadas, permitindo maior controle sobre a utilização de recursos tais como UCP e memória.

6.1.6.1 Executores Monolíticos e Segmentados

Os executores de componentes, identificados através da construção CIDL `manages`, podem ser classificados como monolíticos ou segmentados. Um segmento é um subconjunto das facetas de um componente. Executores monolíticos consideram todas as facetas do componente agrupadas em um único segmento. Os executores segmentados, por sua vez, exigem a presença de mais de um segmento.

Quando uma requisição é feita a uma faceta de um componente, apenas o segmento ao qual esta pertence será ativado. Dessa forma, os executores segmentados permitem melhor utilização de recursos como UCP e memória.

A definição de segmentos, em CIDL, é realizada através da construção `segment`, em negrito na declaração a seguir:

```
composition <category> <composition_name> {
   home executor <home_executor_name> {
      implements < home_type>;
      bindsTo <abstract_storage_home>;
      manages <executor_name> {
         segment <segment_name0> {
            storedOn <abstract_storage_home>;
            provides ( <facet_name0> , <facet_name1> , ... );
         };
         segment <segment_name1> { ... };
         ...
      };
   };
};
```

Cada segmento pode ter sua própria declaração de estado, mapeada através da construção `stores` em um armazenamento abstrato de *homes*. A construção `provides` define quais as facetas que compõem um determinado segmento.

Um componente segmentado possui, ainda, o “segmento do componente”, do qual fazem parte todas as facetas não declaradas em outros segmentos, bem como as demais portas, atributos e interfaces admitidas por herança.

6.1.6.2 Proxy Homes

Um componente CORBA é ativado de modo co-locado em relação ao *home* que o gerencia, ambos no mesmo contêiner. Pode haver um conjunto de operações heterodoxas em um *home* cuja execução independa das instâncias dos componentes, como uma operação que devolva o número de instâncias criadas, por exemplo. Em ambientes distribuídos, pode-se ganhar em escalabilidade se tais operações estiverem “próximas”, como em uma rede local, ao cliente que as utiliza, o que é possível através dos *proxy homes*, como ilustrado na Figura 6.3. As demais operações podem ser delegadas ao *home* ou ainda para o armazenador de *homes*, no caso de componentes que possuam estado persistente.

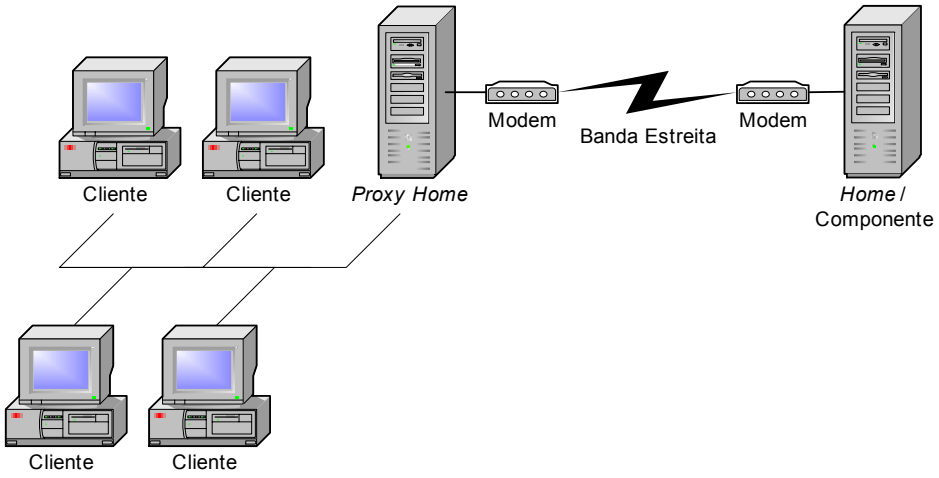

Figura 6.3: utilização de *proxy home* para aumento de escalabilidade.
Nesta configuração se reduz o número de acessos a baixa velocidade.

Outra utilização dos *proxy homes* é no balanceamento de carga, com a delegação de operações para *homes* de acordo com algum critério de balanceamento, conforme ilustrado pela Figura 6.4.

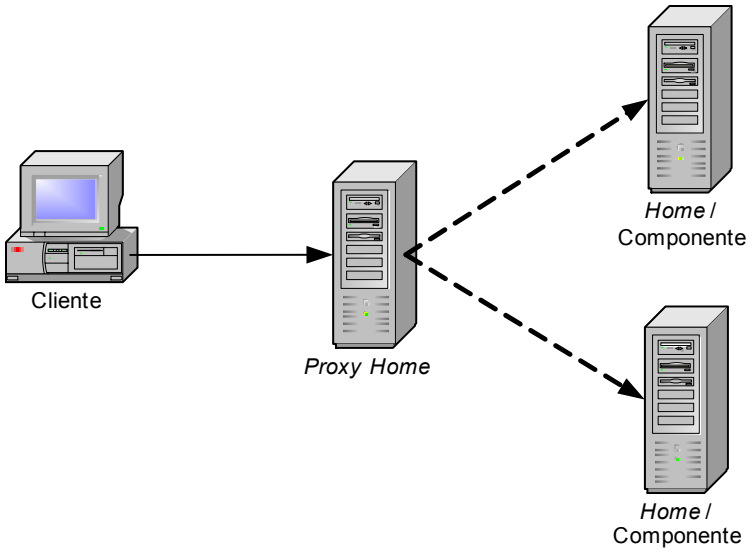

Figura 6.4: utilização de *proxy home* para balanceamento de carga.
As requisições dos clientes são delegadas a um dos *homes*, conforme for conveniente.

A definição de um *proxy home* é realizada através da construção `proxy home` em CIDL, destacada em negrito na declaração:

```
composition <category> <composition_name> {
   home executor <home_executor_name> {
      implements < home_type>;
      bindsTo <abstract_storage_home>;
      manages <executor_name>;
   };
   proxy home <proxy_executor_name> {
      delegatesTo home ( <home_op0>, <home_op1>, ... );
      abstract ( <home_op2>, <home_op3>, ...);
   };
};
```

A construção `delegatesTo home` indica quais operações do *proxy home* serão repassadas (delegadas) ao *home*. A construção `abstract`, do mesmo modo, indica as operações que serão delegadas ao gerenciador abstrato de *homes*.

Capítulo 7

Empacotamento e Distribuição

Em sistemas distribuídos, componentes podem ser implantados em diversos servidores e sistemas operacionais. Além disso, um componente pode depender de outros componentes, tornando o processo de empacotamento e implantação bem complicado.

As implementações de componentes podem ser empacotadas e implantadas. Um pacote (*package*) representa uma ou mais implementações de um componente abstrato, e pode ser instalado em um computador ou agrupado com outros componentes, formando um pacote de montagem (*assembly package*).

Um pacote é constituído por um descritor e um conjunto de arquivos, sendo que o primeiro contém as características do pacote e aponta para seus arquivos, que podem ou não estar dentre os arquivos do pacote que contém o descritor. A Figura 7.1 apresenta a estrutura de diretórios e os arquivos relacionados à implantação do exemplo descrito na seção 3.1.

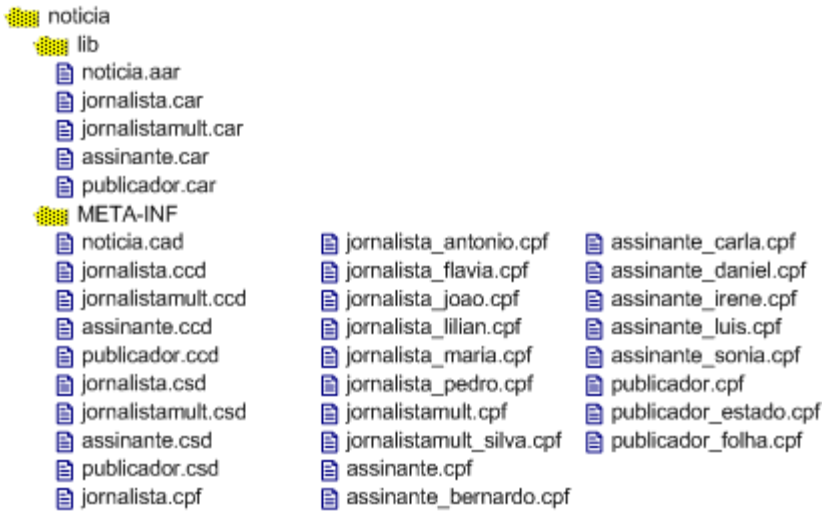

Figura 7.1: estrutura de diretórios e arquivos do exemplo da seção 3.1

Esta estrutura de pacotes utilizada pelo CCM descreve componentes e suas dependências usando *Open Software Description (OSD)*, que é um *XML Document Type Definition (DTD)* proposto ao consórcio WWW por Marimba e Microsoft. Componentes são empacotados em DLLs. Descritores de pacote (*package descriptors*) são documentos XML em conformidade com o OSD DTD, descrevendo o conteúdo da DLL e suas dependências.

CCM OSD também define descritores de montagem (*component assembly descriptors*), que descrevem instruções de implantação e topologia dos componentes, e objetiva dar suporte a implantação automática. Esse descritor especifica ainda as conexões entre componentes, dadas pelas portas `provides`/`uses` e `emits`/`consumes`, e é utilizado como entrada em ferramentas de implantação.

7.1 Empacotamento[11]

O empacotamento de aplicações componentizadas envolve a geração de um conjunto de arquivos, que podem ser divididos em três categorias:

- Arquivos contendo a implementação dos componentes, que são os arquivos .class em Java;
- Arquivos descritores: são arquivos XML, gerados automaticamente, seja através do compilador de CIDL, seja pela ferramenta de empacotamento;
- Arquivos de pacotes, no formato ZIP, contendo os arquivos anteriores.

[11] As extensões aqui apresentadas são sugestões da especificação, não sendo portanto obrigatórias.

A Figura 7.2 apresenta um resumo da relação entre os diversos tipos de arquivos necessários para a implantação.

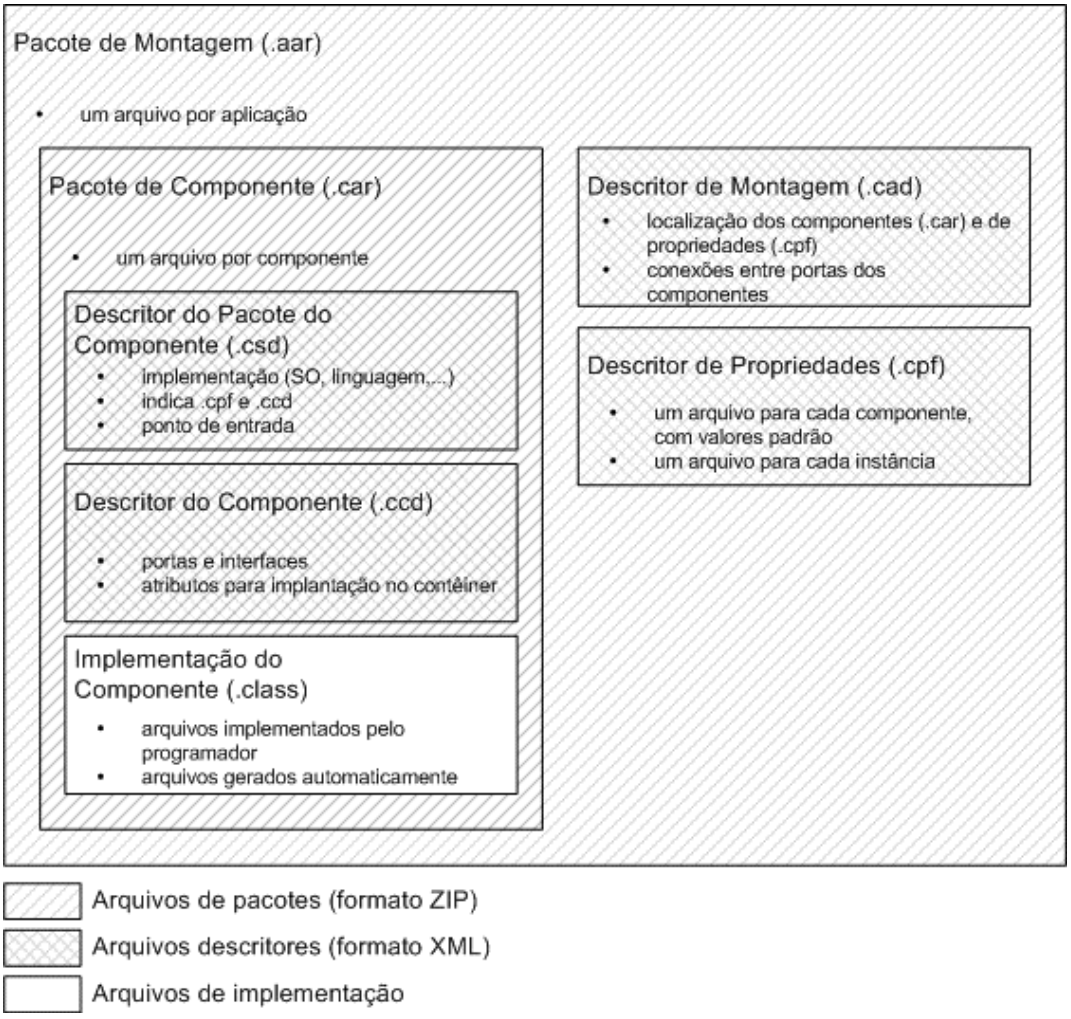

Figura 7.2: esquema básico de empacotamento de Componentes CORBA

7.1.1 Arquivos Descritores

Apesar desses arquivos serem gerados automaticamente, podem ser alterados manualmente pelo usuário. Existem vários descritores envolvidos na implantação de Componentes CORBA, descritos a seguir.

7.1.1.1 Arquivo Descritor de Propriedades (.cpf)

Esse arquivo apresenta os valores dos atributos de cada instância do componente ou de seu *home*, a serem assumidos durante a fase de configuração. Pode haver um arquivo por instância e também um arquivo

adicional, com valores padrão a assumir. Por exemplo[12], o jornalista de nome "Pedro" é configurado no arquivo "jornalista_pedro.cpf":

```
<?xml version="1.0"?>
<!DOCTYPE properties SYSTEM "../dtd/properties.dtd">
<properties>
  <simple name="nome" type="string">
    <description>Nome do Jornalista</description>
    <value>Pedro</value>
    <defaultvalue>Pedro</defaultvalue>
  </simple>
  <simple name="x" type="long">
    <description>Posição horizontal da janela</description>
    <value>779</value>
    <defaultvalue>779</defaultvalue>
  </simple>
  <simple name="y" type="long">
    <description>Posição vertical da janela</description>
    <value>385</value>
    <defaultvalue>385</defaultvalue>
  </simple>
</properties>
```

7.1.1.2 Arquivo Descritor de Componente (.ccd)

Deve haver um arquivo para cada componente, e contém informações para implantação do componente no contêiner, como relação com os serviços de CORBA (segurança e transações, por exemplo), características do componente, como comportamento *multithread*, identificação de funcionalidades no repositório de interface, como para facetas, receptáculos e interfaces admitidas por herança, categoria de componente (seção 5.1.4, "Categorias de Componentes"), política para o ciclo de vida dos serventes (seção 5.2.1.3, "Gerenciamento do Ciclo de Vida") e políticas de acesso a segmentos. Por exemplo, o componente "Jornalista" possui o descritor "jornalista.ccd":

```
<?xml version="1.0"?>
<!DOCTYPE corbacomponent SYSTEM "../dtd/corbacomponent.dtd">
<corbacomponent>
  <corbaversion> 3.0 </corbaversion>
  <componentrepid repid="IDL:Noticia/Jornalista:1.0" />
  <homerepid repid="IDL:Noticia/JornalistaHome:1.0" />
  <componentkind>
    <session>
      <servant lifetime="component" />
    </session>
  </componentkind>
  <security rightsfamily="CORBA" rightscombinator="secanyrights" />
  <threading policy="multithread" />
  <configurationcomplete set="true" />

  <segment name="jornalistaseg" segmenttag="1">
    <segmentmember facettag="1" />
```

[12] Os exemplos de código deste capítulo referem-se àquele descrito no Capítulo 3.

```
    <containermanagedpersistence>
      <storagehome id="PSDL:Noticia/JornalistaHome:1.0" />
      <pssimplementation id="OpenORB-PSS" />
      <catalog type="PSDL:Noticia/JornalistaCatalog:1.0" />
      <accessmode mode="READ_ONLY" />
      <psstransaction policy="TRANSACTIONAL" >
        <psstransactionisolationlevel level="SERIALIZABLE" />
      </psstransaction>
      <params>
        <param name="x" value="1" />
      </params>
    </containermanagedpersistence>
  </segment>

  <homefeatures name="JornalistaHome"
    repid="IDL:Noticia/JornalistaHome:1.0">
  </homefeatures>

  <componentfeatures name="Jornalista" repid="IDL:Noticia/Jornalista:1.0">
    <supportsinterface repid="IDL:Noticia/Identificacao:1.0" />
    <ports>
      <uses
        usesname="recep_publicacao"
        repid="IDL:Noticia/Publicacao:1.0" />
    </ports>
  </componentfeatures>

  <interface name="Identificacao" repid="IDL:Noticia/Identificacao:1.0"/>
  <interface name="Publicacao" repid="IDL:Noticia/Publicacao:1.0"/>
</corbacomponent>
```

A combinação do elemento `componentkind` (com o valor `unclassified`) e de seu elemento filho `poapolicies` permitem a personalização de um conjunto de políticas diferente do utilizado para as categorias descritas na seção 5.1.4, “Categorias de Componentes”, caracterizando a categoria de componente “vazia”.

7.1.1.3 Arquivo Descritor de Montagem (.cad)

Apenas um arquivo por aplicação, apresenta informações necessárias para a implantação de componentes: localização do pacote (arquivo ZIP) contendo a implementação dos componentes, nome e localização do *home* e de cada componente, bem como do arquivo descritor de propriedades padrão, nome de cada instância do componente e do arquivo descritor de propriedades correspondente, conexões de receptáculos a facetas e de consumidores a fontes de eventos. O código abaixo ilustra trechos do descritor “noticia.cad”:

```
<?xml version="1.0"?>
<!DOCTYPE componentassembly SYSTEM "../dtd/componentassembly.dtd">

<componentassembly id="noticia_assembly_descriptor">
  <description>Descritor de Montagem para a aplicação Noticia</description>
  <componentfiles>
    <componentfile id="Jornalista">
```

```
      <fileinarchive name="jornalista.car"/>
    </componentfile>
    <componentfile id="JornalistaMult">
      <fileinarchive name="jornalistamult.car"/>
    </componentfile>
    <componentfile id="Publicador">
      <fileinarchive name="publicador.car"/>
    </componentfile>
    <componentfile id="Assinante">
      <fileinarchive name="assinante.car"/>
    </componentfile>
  </componentfiles>

  <partitioning>
    <hostcollocation cardinality="1">
      <usagename>Execução co-locada</usagename>
      <processcollocation cardinality="1">
        <homeplacement id="JornalistaHome">
          <usagename>Home para componentes do tipo Jornalista</usagename>
          <componentfileref idref="Jornalista"/>
          <componentimplref idref="Jornalista_impl_tie"/>
          <componentproperties>
            <fileinarchive name="META-INF/jornalista.cpf"/>
          </componentproperties>
          <registerwithhomefinder name="JornalistaHome"/>
          <registerwithnaming
name="corbaname:rir:/NameService#EJCCM/JornalistaHome"/>
          <componentinstantiation id="Antonio">
            <usagename>Criação de instância de componente do tipo
Jornalista</usagename>
            <componentproperties>
              <fileinarchive name="META-INF/jornalista_antonio.cpf"/>
            </componentproperties>
            <registercomponent>
              <registerwithnaming name="corbaname:rir:/NameService#Antonio"/>
            </registercomponent>
          </componentinstantiation>

      ...
            <destination>corbaloc::1.2@localhost:2000/NameService</destination>
        </homeplacement>
        <homeplacement id="PublicadorHome">
          <usagename>Home para componentes do tipo Publicador</usagename>
          <componentfileref idref="Publicador"/>
          <componentimplref idref="Publicador_impl_tie"/>
          <componentproperties>
            <fileinarchive name="META-INF/publicador.cpf"/>
          </componentproperties>
          <registerwithhomefinder name="PublicadorHome"/>
          <registerwithnaming
name="corbaname:rir:/NameService#PublicadorHome"/>
          <componentinstantiation id="Estado">
            <usagename>Criação de instância de componente do tipo
Publicador</usagename>
            <componentproperties>
              <fileinarchive name="META-INF/publicador_estado.cpf"/>
            </componentproperties>
            <registercomponent>
              <registerwithnaming name="corbaname:rir:/NameService#Estado"/>
            </registercomponent>
          </componentinstantiation>
      ...
            <destination>corbaloc::1.2@localhost:2000/NameService</destination>
```

```
        </homeplacement>
        <homeplacement id="AssinanteHome">
          <usagename>Home para componentes do tipo Assinante</usagename>
          <componentfileref idref="Assinante"/>
          <componentimplref idref="Assinante_impl_tie"/>
          <componentproperties>
            <fileinarchive name="META-INF/assinante.cpf"/>
          </componentproperties>
          <registerwithhomefinder name="AssinanteHome"/>
          <registerwithnaming name="corbaname:rir:/NameService#AssinanteHome"/>
          <componentinstantiation id="Bernardo">
            <usagename>Criação de instância de componente do tipo
Assinante</usagename>
            <componentproperties>
              <fileinarchive name="META-INF/assinante_bernardo.cpf"/>
            </componentproperties>
            <registercomponent>
              <registerwithnaming name="corbaname:rir:/NameService#Bernardo"/>
            </registercomponent>
          </componentinstantiation>
      ...
            <destination>corbaloc::1.2@localhost:2000/NameService</destination>
        </homeplacement>
    ...
      </processcollocation>
      <extension class="host" origin="ejccm" extra="lyman.cpi.com" />
    </hostcollocation>
  </partitioning>

  <connections>
    <connectinterface>
      <usesport>
        <usesidentifier>recep_publicacao</usesidentifier>
        <componentinstantiationref idref="Antonio"/>
      </usesport>
      <providesport>
        <providesidentifier>faceta_publicacao</providesidentifier>
        <componentinstantiationref idref="Estado"/>
      </providesport>
    </connectinterface>
    ...
    <connectevent>
      <consumesport>
        <consumesidentifier>consumidor_noticia</consumesidentifier>
        <componentinstantiationref idref="Bernardo"/>
      </consumesport>
      <publishesport>
        <publishesidentifier>fonte_noticia</publishesidentifier>
        <componentinstantiationref idref="Estado"/>
      </publishesport>
    </connectevent>
    ...
  </connections>
</componentassembly>
```

7.1.1.4 Arquivo Descritor de Pacote de Componente (.csd)

Um arquivo por componente, contendo detalhes de implementação a serem utilizados para disponibilização, como sistemas operacionais e processador aceitos, bem como compilador e linguagem utilizados. Outras informações

contidas neste arquivo são a indicação da localização dos arquivos de propriedades e descritor do componente, e também o ponto de entrada do componente.

Em inglês, a especificação do CCM [CCM02] se refere a este arquivo como "*Software Package Descriptor*" ao invés de "*Component Package Descriptor*". Isso se deve ao fato do modelo de empacotamento seguir o padrão, e também a nomenclatura, propostos no OSD do consórcio WWW.

Por exemplo, o código abaixo ilustra o descritor de pacote do componente publicador:

```
<?xml version="1.0"?>
<!DOCTYPE softpkg SYSTEM "../dtd/softpkg.dtd">

<softpkg name="Publicador" version="1,0,0,0">
  <pkgtype>CORBA Component</pkgtype>
  <title>Publicador</title>
  <author>
    <name>Alexandre Ricardo Nardi</name>
    <company>IME - USP</company>
    <webpage href="http://www.ime.usp.br/~nardi"/>
  </author>
  <description>Aluno de mestrado</description>
  <license href="http://www.ejccm.org/license.html"/>
  <idl id="IDL:Noticia/Publicador:1.0">
    <link href="../src/idl/noticia.idl"/>
  </idl>
  <descriptor type="CORBA Component">
    <fileinarchive name="META-INF/publicador.ccd"/>
  </descriptor>

  <propertyfile>
    <fileinarchive name="META-INF/publicador.cpf"/>
  </propertyfile>

  <implementation id="Publicador_impl_tie">
    <os name="WinNT" version="4,0,0,0"/>
    <os name="Linux" version="2,2,17,0"/>
    <processor name="x86"/>
    <compiler name="JDK"/>
    <programminglanguage name="Java"/>
    <dependency type="Java Class" action="install">
      <valuetypefactory
        repid="IDL:Noticia/NoticiaEvent:1.0"
        valueentrypoint="Noticia.NoticiaEventDefaultFactory.create"
        factoryentrypoint="Noticia.NoticiaEventDefaultFactory">
        <fileinarchive name="Noticia/NoticiaEventDefaultFactory.class"/>
      </valuetypefactory>
    </dependency>
    <dependency type="ORB" action="assert">
      <name>OpenORB</name>
    </dependency>
    <dependency type=".jar" action="assert">
      <localfile name="ejccm.jar"/>
    </dependency>
    <code type="Java class">
      <fileinarchive name="Noticia/PublicadorHome_impl_tie.class"/>

<entrypoint>Noticia.PublicadorHome_impl_tie.create_home_executor</entrypoint>
```

```
    </code>
    <runtime name="Java VM" version="1,2,2,0"/>
    <runtime name="Java VM" version="1,3,0,0"/>
  </implementation>
</softpkg>
```

7.1.2 Arquivos de Pacotes

7.1.2.1 Pacote de Componente

Contém os arquivos envolvidos na implementação de um componente (com extensão .class, em Java, por exemplo), bem como os descritores do componente (.ccd) e do pacote do componente (.csd).

Esse arquivo não é obrigatório. Entretanto, propicia melhor organização dos arquivos necessários à implantação.

A implementação do CCM aqui empregada [EJCCM03] utiliza a extensão .car para este arquivo.

7.1.2.2 Pacote de Montagem (.aar)

Contém, para cada componente, todos os arquivos do arquivo de pacote de componente ou o próprio arquivo de pacote de componente (.car), sendo este o caso da implementação aqui utilizada [EJCCM03][13]. Inclui, ainda, os arquivos descritores de propriedades (.cpf) e de montagem (.cad).

7.2 Distribuição e Implantação

Uma vez gerados os arquivos descritos anteriormente, estes são distribuídos para os equipamentos onde serão implantados.

Após a distribuição, que pode ser realizada por diversas ferramentas para esse fim, tem início o processo de implantação. Cada implementação do CCM deve fornecer ferramenta ou mecanismo para a implantação de componentes e *homes*.

A especificação do CCM prevê um conjunto de interfaces a serem utilizadas por tal ferramenta na condução da implantação.

O processo de implantação pode ser resumido pela seqüência de passos abaixo:

1. A ferramenta de implantação pede informações ao usuário sobre locais de instalação. Essas informações são armazenadas em uma cópia do arquivo de pacote de montagem;

[13] Portanto, os arquivos .car listados no diretório lib na Figura 7.1 não precisam estar lá, uma vez que estão inclusos no arquivo .aar.

2. A partir dessas informações, a ferramenta de implantação instala cada implementação de componente na plataforma de destino;
3. A ferramenta de implantação cria então uma instância do servidor de componentes (semelhante aos servidores de aplicação do EJB) para a criação do contêiner, dentro do qual será instalado o *home* do componente;
4. O *home* então é utilizado para criar instância de cada componente;
5. Quando for o caso, um configurador será aplicado a cada componente;
6. Com todos os componentes instalados e configurados, são estabelecidas as conexões entre os componentes (receptáculos a facetas e para eventos);
7. A operação `configuration_complete` será chamada para cada componente.

O processo de implantação de Componentes CORBA deve ser o mais simples possível para o usuário. Dessa forma, a tarefa de disponibilizar novas versões de componentes não demanda a presença de especialistas, podendo ser realizada por profissionais que não possuam perfil de desenvolvedor.

A implementação empregada neste estudo [EJCCM03], por exemplo, utiliza a ferramenta Ant [ANT03] para a implantação e execução de aplicações componentizadas. A criação de nova aplicação requer poucas intervenções por parte do usuário, realizadas em um único lugar: o arquivo `build.xml`. Outros produtos podem exigir ainda menos esforço, por meio de ferramentas gráficas.

O trecho de código a seguir ilustra os segmentos do arquivo `build.xml` que foram acrescidos no intuito de implantar e executar a aplicação do exemplo descrito na seção 3.1:

```
<!--
  ===================================================
    EJCCM Configuration
  ===================================================
-->
<property name="demo.noticia.dir" location="${demo.dir}/Noticia" />
<property name="build.demo.noticia.dir" location="${build.demo.dir}/Noticia" />


<!--
  ===================================================
    Run noticia demo
  ===================================================
-->
<target name="run-demo-noticia" description="run the noticia demo">
   <parallel>
        <sequential>
           <echo message="Running the noticia demo ..." />
           <java fork="true" failonerror="true"
             dir="${build.exec.dir}"
```

```
            classname="ejccm.Components.Deployment.Deploy">
             <sysproperty key="openorb.debug" value="0" />
             <sysproperty key="org.omg.CORBA.ORBSingletonClass"
               value="org.openorb.CORBA.ORBSingleton" />
             <sysproperty key="org.omg.CORBA.ORBClass"
               value="org.openorb.CORBA.ORB" />
             <sysproperty key="openorb.config"
               value="${config.dir}/OpenORB.xml" />
             <classpath>
                <pathelement location="${ejccm.jar}" />
                <path refid="orb.all.classpath" />
                <path refid="tools.classpath" />
             </classpath>
             <arg line="-cad-file
               ${demo.noticia.dir}/lib/demo_noticia.aar -name-
               service ${nameservice} -debug -install -build -
               tear-down -pause 3600" />
          </java>
        </sequential>
       </parallel>
</target>
```

Capítulo 8

Considerações Finais

O desenvolvimento de sistemas distribuídos traz consigo um conjunto de questões, como localização remota de objetos, comportamento transacional, persistência e escalabilidade. Esses e outros fatores tornam o trabalho do desenvolvedor mais complexo, desviando sua atenção, que deveria ser concentrada na resolução de problemas de negócio.

Arcabouços como o CCM procuram simplificar o processo de análise e, principalmente, de desenvolvimento dos sistemas distribuídos. Tal simplificação, conforme vimos neste texto, é obtida por intermédio de funcionalidades como a geração automática de artefatos pelo CIF, organização em portas dos recursos fornecidos e consumidos pelos componentes, presença de um grau a mais de abstração entre o componente e o ORB, dado pelos *homes* e contêineres e estruturação do processo de empacotamento e distribuição de soluções componentizadas. Ao mesmo tempo, o CCM apresenta os benefícios de CORBA, como os diversos serviços disponíveis, sendo genérico e adequado a ambientes heterogêneos.

Atualmente existem algumas implementações disponíveis, que podem ser localizadas em [Ruiz01], cada qual com suas características, tais como a linguagem de implementação e os recursos do CCM disponibilizados. Esse fato é positivo, uma vez que demonstra o interesse do mercado e do ambiente acadêmico por esta solução, abrindo possibilidades interessantes, principalmente em ambientes heterogêneos, uma vez que outros arcabouços, como o EJB e o Microsoft.NET, possuem base sólida em ambientes específicos, no que concerne a linguagens e plataformas.

Além disso, cremos que há espaço para o CCM em um outro nicho: o das aplicações CORBA atualmente existentes. A componentização destas

aplicações, ou de novas versões destas, propicia um grau a mais de abstração, conforme apresentado neste texto.

Pesquisas têm sido realizadas em pontos específicos, como o uso de Componentes CORBA em aplicações de alto desempenho e de tempo real [WLS00], a extensão das implementações OpenCCM [GOAL01] e MicoCCM [FPX01] para utilização paralela de componentes [PPR02], e a extensão do OpenCCM para tratar o gerenciamento de replicação de componentes [MH02]. Essa movimentação, bem como o resultado das pesquisas, sempre positivo, sugerem o uso de soluções componentizadas, em particular com o CCM.

O sítio em [Ruiz01] concentra diversas informações sobre o CCM, incluindo artigos, apresentações e referências para implementações. Há também, neste sítio, uma lista de discussão, à qual recorremos algumas vezes durante os testes que realizamos.

A receptividade inicial do CCM foi excelente, conforme a opinião [RR99] de diversos especialistas, de empresas como IONA Technologies, Sun Microsystems, Fujitsu e Unysis, entre outras.

Contudo, a morosidade da OMG no processo de discussão e aprovação da especificação do CCM representa um grande desafio para os implementadores de ORBs: produzir um produto que possa competir com similares no mercado. Foram cinco anos, desde a publicação da versão final da RFP para o CCM [RFP97], em 1997, até a versão final da especificação do CCM, em 2002. Durante esse tempo, foi grande a evolução de outras tecnologias voltadas para sistemas distribuídos, como Microsoft.NET e acréscimos em novas versões de EJB, o que reduz significativamente as chances de adoção do CCM pelo mercado de informática como um todo.

Além disso, há ainda que ser analisado o impacto que a utilização de *Web Services* e a evolução de suas especificações [W3C03, WSI03] terão sobre outras tecnologias para sistemas distribuídos, como o CCM.

Por fim, apesar das dificuldades acima, o CCM vem ao encontro dos anseios de diversos desenvolvedores com os quais mantivemos contato durante a realização deste trabalho, principalmente quanto à simplificação e melhor organização do processo de desenvolvimento de sistemas distribuídos.

8.1 Trabalhos Futuros

Acreditamos que este trabalho possa colaborar no sentido de apresentar as funcionalidades e os benefícios do CCM, seja para o leitor que deseja conhecer uma visão geral do arcabouço, seja para o que efetivamente deseja desenvolver sistemas componentizados.

Ainda no sentido de facilitar o uso do CCM pelo programador, outros trabalhos podem ser desenvolvidos, principalmente a partir de implementações mais completas e estáveis. Deixamos aqui as seguintes sugestões:

- Elaboração de um tutorial que leve o desenvolvedor através do processo de construção de uma aplicação simples. Para este tutorial pode ser utilizado, como ponto de partida, um dos exemplos que acompanham a implementação escolhida do CCM, ou ainda o que desenvolvemos neste trabalho;
- Desenvolvimento de um guia de referência, com exemplos objetivos e que possam ser implementados e executados, para cada funcionalidade do CCM, como uso das diversas portas, atributos, fase de configuração de componentes, funcionalidades associadas aos tipos de composições e integração com os serviços de CORBA.

Apêndice A

Outras Tecnologias para Componentes

A especificação do CCM aborda o desenvolvimento de sistemas componentizados, assim como outras tecnologias o fazem. Neste apêndice apresentaremos algumas dessas tecnologias, traçando comparações com o CCM.

A.1 Enterprise Java Beans

A proposição do CCM foi realizada como um superconjunto da especificação do EJB 1.1 [EJB03]. Assim, novos recursos foram acrescidos, como:

- Diversos tipos de portas: a especificação do EJB não prevê facetas, receptáculos ou portas para eventos;
- Construções adicionais para categorizar componentes: EJB prevê apenas *session* e *entity beans*, enquanto CCM possibilita o uso de componentes de serviço e processo, além de sessão e entidade;
- Herança explícita de componentes, de modo visível ao usuário;
- Facilidades decorrentes da CIF.

Por outro lado, a especificação EJB evoluiu [EJB03], com funcionalidades que não foram contempladas na especificação do CCM, tais como:

- *Message-driven beans*: um novo tipo de componente, acessado pelos cliente apenas indiretamente, através de mensagens enviadas ao JMS, *Java Message Service*;

- Além do contêiner gerenciar estado para os *entity beans*, este pode agora gerenciar os relacionamentos entre eles;
- Definição da *Enterprise Java Beans Query Language*, ou EJB QL, uma linguagem declarativa para implementação de métodos de pesquisa a serem oferecidos pelo contêiner.

Deste modo, CCM não pode ser considerado genericamente como um superconjunto de EJB, mas apenas de EJB 1.1.

A.2 Microsoft .NET

A plataforma .NET [NET03] é um conjunto de tecnologias que permite conectividade entre sistemas e aplicações, seja em uma rede local ou ainda através da Internet, podendo se utilizar de *web services* [W3C03]. O principal integrante da plataforma é o arcabouço .NET (.NET *Framework*), que contém um conjunto de bibliotecas básicas e o *Common Language Runtime*, com finalidade comparável à da JVM. Diferentemente do que ocorria antes do lançamento da plataforma .NET, a Microsoft abriu a especificação do arcabouço, de modo que fornecedores independentes possam desenvolver versões específicas do mesmo para suas plataformas, a exemplo do que ocorreu com o projeto Mono [MONO03], que fornece uma versão do arcabouço para Linux.

Essa definição para a plataforma .NET mostra que não é possível uma comparação direta entre CCM, ou mesmo EJB, e .NET. A literatura existente relata, por outro lado, diversos textos e exemplos de código comparando J2EE com .NET.

Todavia, .NET prevê o desenvolvimento de componentes, e estes possuem as seguintes características:

- Não é necessário utilizar MIDL, *Microsoft Interface Definition Language*, o que se deve a dois fatores:
  - O resultado da compilação de programas em qualquer linguagem que siga a especificação prevista na plataforma .NET, denominada *Common Language Specification*, ou CLS, é um código expresso em uma linguagem intermediária, a *Microsoft Intermediate Language* - MSIL. Esse código é convertido em binário no momento da execução;
  - Funcionalidades dos componentes podem ser expostas a outros sistemas por meio de *web services*, que seguem padrões específicos.
- A Microsoft recomenda, quando necessário, o uso de recursos do sistema operacional Windows 2000, XP ou 2003, relativos a servidor de aplicações. Exemplos desses recursos são o monitor transacional e o mecanismo de segurança embutidos no COM+. Para utilizar

esses recursos, componentes .NET são encapsulados em componentes COM [BOX98, COM03];

- Assim como os componentes COM, os componentes .NET permitem a definição de interfaces análogas às facetas dos componentes CORBA. Entretanto, não há equivalente em .NET para os outros tipos de portas, como receptáculos e eventos;
- O papel de contêiner é desempenhado pelo arcabouço .NET e pelo COM+, quando for o caso.

Assim como os componentes CORBA dependem da existência de um ORB específico para a plataforma em que se deseja executá-lo, os EJBs dependem da JVM e os componentes .NET dependem do arcabouço .NET.

Apêndice B

Integração com EJB

A especificação do CCM foi desenvolvida levando em consideração diversas funcionalidades presentes na especificação do *Enterprise Java Beans 1.1* [Thom98]. De fato, CCM é um super conjunto desta versão do EJB, possuindo o mesmo conceito de componentes que são executados em contêineres. Aliado a isso, a base instalada em EJB justifica a integração das duas tecnologias. Outro fator que torna natural a escolha de EJB como tecnologia a integrar com o CCM é a utilização de CORBA pelo EJB. É possível, ainda integrar CCM com outras tecnologias, como COM/DCOM, porém exige maior esforço na construção de pontes (*bridges*) para tal.

A integração CCM/EJB segue o mecanismo descrito na especificação de CORBA 3 [CORBA3, capítulo 17, "*Interworking Architecture*"]. Em linhas gerais, é necessária a presença de uma ponte entre os dois sistemas, atuando de modo que um objeto de um sistema A é visto pelo sistema B como se fosse um objeto do sistema B. À forma como um objeto de um sistema enxerga um objeto de outro sistema dá-se o nome de visão. O modelo básico de uma ponte é ilustrado pela Figura B.1.

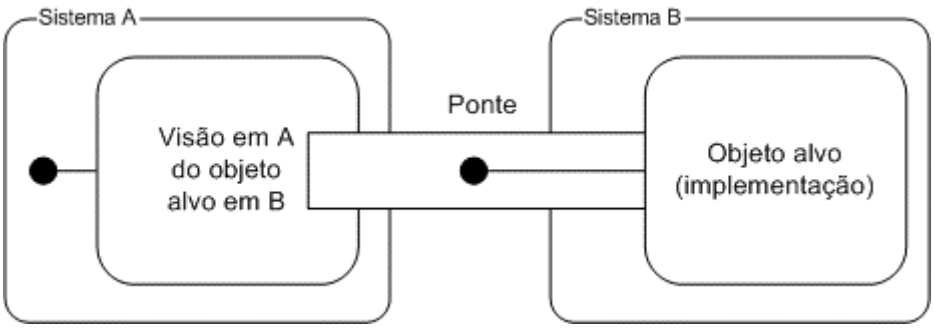

Figura B.1: modelo de uma ponte

No caso da integração CCM/EJB, podem ocorrer duas situações:

1. Um componente CCM sendo visto por um cliente EJB através de uma visão EJB de um componente CORBA;
2. Um componente EJB sendo visto por um cliente CCM através de uma visão CCM de um componente EJB.

Essas possibilidades estão representadas na Figura B.2.

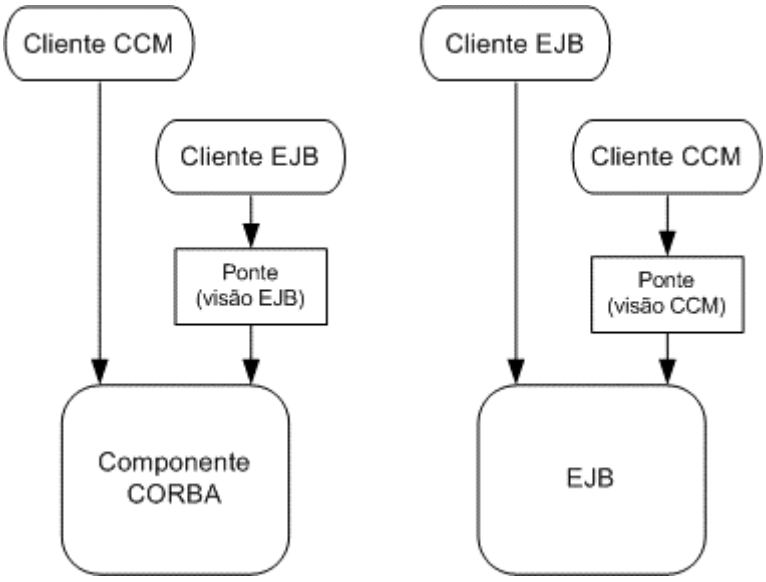

Figura B.2: integração CCM/EJB

B.1 Visões CCM para EJB

Permitem que um cliente CORBA utilize uma visão CCM para acessar um EJB (como se este fosse um Componente CORBA). O exemplo abaixo ilustra a interface de um EJB e como este é exposto (em IDL) ao cliente CCM:

Interface do EJB

```
package exemplo;

class RemedioInfo implements java.io.Serializable
{
       public int codigo;
       public String nome;
       public String descricao;
}

interface Remedio extends javax.ejb.EJBObject
{
       RemedioInfo getRemedioInfo(int codigo) throws java.rmi.RemoteException;
}

interface RemedioHome extends javax.ejb.EJBHome
{
       Remedio create() throws java.rmi.RemoteException;
}
```

Visão CCM do EJB

```
valuetype RemedioInfo {
       public long codigo;
       public ::CORBA::WStringValue nome;
       public ::CORBA::WStringValue descricao;
};

interface RemedioDefault {
       RemedioInfo getRemedioInfo(in long codigo);
};

component Remedio supports RemedioDefault {};

home RemedioHome manages Remedio {
       factory create();
};
```

B.2 Visões EJB para Componentes CORBA

Permitem que um cliente Java (que pode ser um *Java Bean* ou EJB) utilize uma visão EJB para acessar um Componente CORBA (como se este fosse um EJB). Vale notar que qualquer cliente Java pode acessar um Componente CORBA diretamente, sem a necessidade de uma visão EJB. A única diferença é a sintaxe das operações que, não obstante, é muito parecida [CCM02, página 5-18].

O exemplo a seguir ilustra a definição, em IDL, de um componente CORBA e as interfaces para acesso a este como um EJB.

Interface (IDL) do componente CORBA

```
interface Conta {
       void debito(in double valor) raises (SaldoInsuficiente);
       void credito(in double valor);
};

component ContaCorrente supports Conta {
       readonly attribute double saldo;
};

valuetype NumeroConta {
       public long Numero;
};

home ContaCorrenteHome manages ContaCorrente primarykey NumeroConta {
       finder contaImportante(double saldominimo);
};
```

Visão EJB do Componente CORBA

```
public interface ContaCorrente extends javax.ejb.EJBObject {
       public void debito(double valor)
              throws SaldoInsuficiente, java.rmi.RemoteException;
       public void credito(double valor) throws java.rmi.RemoteException;
       public double getSaldo() throws java.rmi.RemoteException;
}

public class NumeroConta implements java.io.Serializable {
       public long Numero;
       public NumeroConta( long k )
       {
              Numero = k;
       }
}

public interface ContaCorrenteHome extends javax.ejb.EJBHome {
       public ContaCorrente create(NumeroConta numero)
              throws DuplicateKeyException, CreateException,
              java.rmi.RemoteException;
       public ContaCorrente findByPrimaryKey(ContaCorrente key )
              throws ObjectNotFoundException, FinderException,
              java.rmi.RemoteException;
       public ContaCorrente findByContaImportante(double saldominimo)
              throws java.rmi.RemoteException;
}
```

Apêndice C

Código-Fonte do Exemplo

Este apêndice apresenta apenas o código implementado pelo programador. A partir dos arquivos abaixo, a implementação do CCM utilizada gera mais de 250 outros artefatos, tornando sua reprodução aqui sem sentido.

Os arquivos para empacotamento e distribuição foram ilustrados no Capítulo 7. Por isso não serão listados neste apêndice.

C.1 Definição dos Componentes em IDL3 – “noticia.idl”

```
/*********************************************************************/
/***  Definição dos componentes em IDL3.
/*********************************************************************/

// O módulo Components contém as definições em IDL para o CCM.
import Components;
#pragma prefix "Noticia"


// Interface a ser fornecida como faceta pelo componente Publicador
// e usada como receptáculo pelo componente Jornalista
interface Publicacao
{
    void publica_noticia(in string texto);
};


// Interface a ser admitida por herança pelos componentes
// cujas instâncias possuam nome. O atributo contido nesta
// interface poderia ser declarado em cada componente.
interface Identificacao
{
```

```
    attribute string nome;
};


// Interface a ser admitida por herança pelos componentes,
// para posicionamento da janela de cada instância.
interface Dimensoes
{
    attribute long x;
    attribute long y;
    attribute long altura;
    attribute long largura;
};


// Event type para evento a ser publicado pelo componente Publicador
// e consumido pelo componente Assinante.
eventtype NoticiaEvent : Components::EventBase
{
    // Este evento contém apenas o texto da notícia a publicar.
    public string texto;
    factory create(in string nome);
};


// ===================================================================
//
// Componente Publicador.
//
// ===================================================================
component Publicador supports Identificacao, Dimensoes
{
    // Disponibiliza faceta para os componentes do tipo Jornalista
    // e JornalistaMult
    provides Publicacao faceta_publicacao;

    // Publica as notícias aos componentes do tipo Assinante
    publishes NoticiaEvent fonte_noticia;
};

// ===================================================================
//
// Home para o componente Publicador.
//
// ===================================================================
home PublicadorHome manages Publicador
{
};


// ===================================================================
//
// Componente Jornalista.
//
// ===================================================================
component Jornalista supports Identificacao, Dimensoes
{
    // Utiliza, através deste receptáculo, a operação
    // para publicação de notícias oferecida por uma instância do
    // componente Publicador
    uses Publicacao recep_publicacao;
};
```

```
// ========================================================================
//
// Home para o componente Jornalista.
//
// ========================================================================
home JornalistaHome manages Jornalista
{
};


// ========================================================================
//
// Componente Jornalista.
//
// ========================================================================
component JornalistaMult supports Identificacao, Dimensoes
{
    // Utiliza, através deste receptáculo, a operação
    // para publicação de notícias oferecida por uma
    // ou mais instâncias do componente Publicador
    uses multiple Publicacao recep_publicacao;
};

// ========================================================================
//
// Home para o componente JornalistaMult.
//
// ========================================================================
home JornalistaMultHome manages JornalistaMult
{
};


// ========================================================================
//
// Componente Assinante.
//
// ========================================================================
component Assinante supports Identificacao, Dimensoes
{
    // Consome eventos emitidos pelas instâncias
    // do componente Publicador
    consumes NoticiaEvent consumidor_noticia;
};

// ========================================================================
//
// Home para o componente Assinante.
//
// ========================================================================
home AssinanteHome manages Assinante
{
};
```

C.2 Definição das Composições em CIDL – “noticia.cidl”

```
/*********************************************************************/
/***  Definição de composições em CIDL.
/*********************************************************************/

#include <noticia.idl>

module Jornalista
{
    module Simples
    {
        composition session JornalistaComposition
        {
            home executor JornalistaHome_impl
            {
                implements JornalistaHome;
                manages Jornalista_impl;
            };
        };
    };

    module Multiplo
    {
        composition session JornalistaMultComposition
        {
            home executor JornalistaMultHome_impl
            {
                implements JornalistaMultHome;
                manages JornalistaMult_impl;
            };
        };
    };
};

module Assinante
{
    composition session AssinanteComposition
    {
        home executor AssinanteHome_impl
        {
            implements AssinanteHome;
            manages Assinante_impl;
        };
    };
};

module Publicador
{
    composition session PublicadorComposition
    {
        home executor PublicadorHome_impl
        {
            implements PublicadorHome;
            manages Publicador_impl;
        };
    };
};
```

C.3 Implementação do Componente Jornalista

```
/*********************************************************************/
/***  Implementação do componente Noticia.Jornalista.
/***  Esse componente possui um receptáculo simplex,
/***  "recep_publicacao", que deve ser conectado à faceta
/***  "faceta_publicacao" de uma instância do componente Publicador.
/***  O jornalista deve descrever a notícia no campo texto, e enviá-la
/***  ao publicador através do receptáculo acima.
/*********************************************************************/

package Noticia;

public class Jornalista_impl_tie
    extends    org.omg.CORBA.LocalObject
    implements org.omg.Components.SessionComponent,
               org.omg.Components.ExecutorLocator,
               CCM_Jornalista_Executor,
               java.awt.event.ActionListener
{
    // ================================================================
    //
    // Estado interno.
    //
    // ================================================================

    // Nome de uma instância do componente
    private String nome_ = "sem nome";

    // Dimensões da janela de uma instância do componente
    private int x_, y_, altura_, largura_;

    // Referências à interface com usuário
    private javax.swing.JFrame frame_;
    private javax.swing.JTextField text_;

    // Referência ao contexto em que o componente é executado
    private CCM_Jornalista_Context contexto_;


    // ================================================================
    //
    // Construtor.
    //
    // ================================================================

    public Jornalista_impl_tie()
    {
    }


    // ================================================================
    //
    // Métodos para a interface SessionComponent.
    //
    // ================================================================

    // Operação acionada na ativação do componente de sessão.
    public void set_session_context(org.omg.Components.SessionContext context)
        throws org.omg.Components.CCMException
    {
        contexto_ = (CCM_Jornalista_Context) context;
```

```
    }

    // Operação chamada pelo contêiner (callback) para informar que o
componente está ativo.
    public void ccm_activate()
        throws org.omg.Components.CCMException
    {
    }

    // Operação chamada pelo contêiner (callback) para informar que o
componente está sendo desativado.
    public void ccm_passivate()
        throws org.omg.Components.CCMException
    {
    }

    // Operação chamada pelo contêiner (callback) para informar que o
componente está sendo destruído.
    public void ccm_remove()
        throws org.omg.Components.CCMException
    {
        // Destrói a interface com o usuário
        frame_.dispose();
        frame_ = null;
    }


    // ================================================================
    //
    // Métodos para a interface ExecutorLocator.
    //
    // ================================================================

    public org.omg.CORBA.Object obtain_executor(java.lang.String nome)
        throws org.omg.Components.CCMException
    {
        if (nome.equals("Jornalista"))
        {
            return this;
        }
        else
        {
            throw new org.omg.Components.CCMException();
        }
    }

    public void release_executor(org.omg.CORBA.Object exc)
        throws org.omg.Components.CCMException
    {
    }

    // Encerra a fase de configuração do componente
    public void configuration_complete()
        throws org.omg.Components.InvalidConfiguration
    {
        // Verifica se a fase de configuração foi concluída.
         if(nome_.equals("sem nome"))
            throw new org.omg.Components.InvalidConfiguration();

         // Verifica se este componente está conectado ao componente publicador.
         if(contexto_.get_connection_recep_publicacao() == null)
            throw new org.omg.Components.InvalidConfiguration();
```

```
    // Inicializa a interface com o usuário.

    // Cria a área principal.
    frame_ = new javax.swing.JFrame(nome_ + "'s Jornalista GUI");
    // Dimensiona a janela.
    frame_.setBounds(x_, y_, largura_, altura_);

    // Cria um campo para o jornalista escrever a notícia.
    text_ = new javax.swing.JTextField("", 1);

    // Cria um botão para envio de notícia ao publicador.
    javax.swing.JButton button = new javax.swing.JButton("Envia Notícia");
    button.addActionListener(this);

    // Cria e apresenta a janela.
    javax.swing.JPanel panel = new javax.swing.JPanel(
                                    new java.awt.BorderLayout());
    frame_.getContentPane().add(panel);
    panel.add(new javax.swing.JScrollPane(text_),
              java.awt.BorderLayout.CENTER);
    panel.add(button, java.awt.BorderLayout.SOUTH);
    frame_.show();
}

protected void finalize()
    throws Throwable
{
    super.finalize();
}


// ====================================================================
//
// Métodos para o executor do componente
// (interface CCM_Jornalista_Executor).
//
// ====================================================================

// Métodos para o atributo "nome"
public void nome(String n)
{
    nome_ = n;
    if (frame_ != null)
        frame_.setTitle(nome_ + "'s Jornalista GUI");
}

public String nome()
{
    return nome_;
}

// Métodos para o atributo "x"
public void x(int valor)
{
    x_ = valor;
    if (frame_ != null)
        frame_.setLocation(x_, y_);
}

public int x()
{
    return x_;
}
```

```
    // Métodos para o atributo "y"
    public void y(int valor)
    {
        y_ = valor;
        if (frame_ != null)
            frame_.setLocation(x_, y_);
    }

    public int y()
    {
        return y_;
    }

    // Métodos para o atributo "altura"
    public void altura(int valor)
    {
        altura_ = valor;
        if (frame_ != null)
            frame_.setSize(largura_, altura_);
    }

    public int altura()
    {
        return altura_;
    }

    // Métodos para o atributo "largura"
    public void largura(int valor)
    {
        largura_ = valor;
        if (frame_ != null)
            frame_.setSize(largura_, altura_);
    }

    public int largura()
    {
        return largura_;
    }


    // ========================================================================
    //
    // Métodos para a interface java.awt.event.ActionListener.
    //
    // ========================================================================

    // Envio de notícia ao publicador
    public void actionPerformed(java.awt.event.ActionEvent evento)
    {
        // Obtém referência à faceta associada a 'recep_publicacao'
        Publicacao faceta = contexto_.get_connection_recep_publicacao();

        // Verifica se a conexão está disponível.
        if(faceta == null)
        {
            System.err.println("Erro na configuração de recep_publicacao");
            return;
        }
        // Envia notícia ao publicador.
        faceta.publica_noticia(nome_ + ":" + text_.getText());
    }
}
```

C.4 Implementação do *Home* do Componente Jornalista

```
/**************************************************************************/
/***  Implementação do home Noticia.JornalistaHome.
/**************************************************************************/

package Noticia;

public class JornalistaHome_impl_tie
    extends    org.omg.CORBA.LocalObject
    implements CCM_JornalistaHome
{
    // ========================================================================
    //
    // Construtor.
    //
    // ========================================================================

    public JornalistaHome_impl_tie()
    {
    }


    // ========================================================================
    //
    // Métodos para a interface CCM_JornalistaHome
    //
    // ========================================================================

    // Cria uma instância do executor do componente.
    public org.omg.Components.EnterpriseComponent create()
    {
        return new Jornalista_impl_tie();
    }


    // ========================================================================
    //
    // Métodos para implantação.
    //
    // ========================================================================

    // Método chamado pelo servidor de componentes
    public static org.omg.Components.HomeExecutorBase create_home_executor()
    {
        return new JornalistaHome_impl_tie();
    }

    protected void finalize()
        throws Throwable
    {
        super.finalize();
    }
}
```

C.5 Implementação do Componente JornalistaMult

```
/*********************************************************************/
/***  Implementação do componente Noticia.JornalistaMult.
/***  Esse componente possui um receptáculo multiplex,
/***  "recep_publicacao", que deve ser conectado à faceta
/***  "faceta_publicacao" de uma ou mais instâncias do
/***  componente Publicador.
/***  O jornalista deve descrever a notícia no campo texto, e enviá-la
/***  aos publicadores conectados através do receptáculo acima.
/*********************************************************************/

package Noticia;

public class JornalistaMult_impl_tie
    extends    org.omg.CORBA.LocalObject
    implements org.omg.Components.SessionComponent,
               org.omg.Components.ExecutorLocator,
               CCM_JornalistaMult_Executor,
               java.awt.event.ActionListener
{
    // ================================================================
    //
    // Estado interno.
    //
    // ================================================================

    // Nome de uma instância do componente
    private String nome_ = "sem nome";

    // Dimensões da janela de uma instância do componente
    private int x_, y_, altura_, largura_;

    // Referências à interface com usuário
    private javax.swing.JFrame frame_;
    private javax.swing.JTextField text_;

    // Referência ao contexto em que o componente é executado
    private CCM_JornalistaMult_Context contexto_;


    // ================================================================
    //
    // Construtor.
    //
    // ================================================================

    public JornalistaMult_impl_tie()
    {
    }


    // ================================================================
    //
    // Métodos para a interface SessionComponent.
    //
    // ================================================================

    // Operação acionada na ativação do componente de sessão.
    public void set_session_context(org.omg.Components.SessionContext context)
        throws org.omg.Components.CCMException
    {
```

```
        contexto_ = (CCM_JornalistaMult_Context) context;
    }

    // Operação chamada pelo contêiner (callback) para informar que o
componente está ativo.
    public void ccm_activate()
        throws org.omg.Components.CCMException
    {
    }

    // Operação chamada pelo contêiner (callback) para informar que o
componente está sendo desativado.
    public void ccm_passivate()
        throws org.omg.Components.CCMException
    {
    }

    // Operação chamada pelo contêiner (callback) para informar que o
componente está sendo destruído.
    public void ccm_remove()
        throws org.omg.Components.CCMException
    {
        // Destrói a interface com o usuário
        frame_.dispose();
        frame_ = null;
    }


    // ================================================================
    //
    // Métodos para a interface ExecutorLocator.
    //
    // ================================================================

    public org.omg.CORBA.Object obtain_executor(java.lang.String nome)
        throws org.omg.Components.CCMException
    {
        if (nome.equals("JornalistaMult"))
        {
            return this;
        }
        else
        {
            throw new org.omg.Components.CCMException();
        }
    }

    public void release_executor(org.omg.CORBA.Object exc)
        throws org.omg.Components.CCMException
    {
    }

    // Encerra a fase de configuração do componente
    public void configuration_complete()
        throws org.omg.Components.InvalidConfiguration
    {
        // Verifica se a fase de configuração foi concluída.
        if(nome_.equals("sem nome"))
            throw new org.omg.Components.InvalidConfiguration();

        // Verifica se este componente está conectado ao componente publicador.
        if(contexto_.get_connections_recep_publicacao() == null)
            throw new org.omg.Components.InvalidConfiguration();
```

```
    // Inicializa a interface com o usuário.

    // Cria a área principal.
    frame_ = new javax.swing.JFrame(nome_ + "'s Jornalista Multiplex GUI");
    // Dimensiona a janela.
    frame_.setBounds(x_, y_, largura_, altura_);

    // Cria um campo para o jornalista escrever a notícia.
    text_ = new javax.swing.JTextField("", 1);

    // Cria um botão para envio de notícia ao publicador.
    javax.swing.JButton button = new javax.swing.JButton("Envia Notícia");
    button.addActionListener(this);

    // Cria e apresenta a janela.
    javax.swing.JPanel panel = new javax.swing.JPanel(
                                    new java.awt.BorderLayout());
    frame_.getContentPane().add(panel);
    panel.add(new javax.swing.JScrollPane(text_),
              java.awt.BorderLayout.CENTER);
    panel.add(button, java.awt.BorderLayout.SOUTH);
    frame_.show();
}

protected void finalize()
    throws Throwable
{
    super.finalize();
}


// ========================================================================
//
// Métodos para o executor do componente
// (interface CCM_JornalistaMult_Executor).
//
// ========================================================================

// Métodos para o atributo "nome"
public void nome(String n)
{
    nome_ = n;
    if (frame_ != null)
        frame_.setTitle(nome_ + "'s Jornalista Multiplex GUI");
}

public String nome()
{
    return nome_;
}

// Métodos para o atributo "x"
public void x(int valor)
{
    x_ = valor;
    if (frame_ != null) frame_.setLocation(x_, y_);
}

public int x()
{
    return x_;
}
```

```
    // Métodos para o atributo "y"
    public void y(int valor)
    {
        y_ = valor;
        if (frame_ != null) frame_.setLocation(x_, y_);
    }

    public int y()
    {
        return y_;
    }

    // Métodos para o atributo "altura"
    public void altura(int valor)
    {
        altura_ = valor;
        if (frame_ != null) frame_.setSize(largura_, altura_);
    }

    public int altura()
    {
        return altura_;
    }

    // Métodos para o atributo "largura"
    public void largura(int valor)
    {
        largura_ = valor;
        if (frame_ != null) frame_.setSize(largura_, altura_);
    }

    public int largura()
    {
        return largura_;
    }


    // ========================================================================
    //
    // Métodos para a interface java.awt.event.ActionListener.
    //
    // ========================================================================

    // Envio de notícia aos publicadores conectados
    public void actionPerformed(java.awt.event.ActionEvent evento)
    {
        // Obtém referência às facetas associadas a 'recep_publicacao'
        Noticia.JornalistaMultPackage.recep_publicacaoConnection[] facetas =
            contexto_.get_connections_recep_publicacao();
        // Verifica se a conexão está disponível.
        if(facetas == null)
        {
            System.err.println("Erro na configuração de recep_publicacao");
            return;
        }
        // Envia notícia a cada publicador conectado.
        for(int i=0; i < facetas.length;i++)
        {
            facetas[i].objref.publica_noticia(nome_ + ":" + text_.getText());
        }
    }
}
```

C.6 Implementação do *Home* do Componente JornalistaMult

```
/*********************************************************************/
/***  Implementação do home Noticia.JornalistaMultHome.
/*********************************************************************/

package Noticia;

public class JornalistaMultHome_impl_tie
    extends    org.omg.CORBA.LocalObject
    implements CCM_JornalistaMultHome
{
    // ====================================================================
    //
    // Construtor.
    //
    // ====================================================================

    public JornalistaMultHome_impl_tie()
    {
    }


    // ====================================================================
    //
    // Métodos para a interface CCM_JornalistaMultHome
    //
    // ====================================================================

    // Cria uma instância do executor do componente.
    public org.omg.Components.EnterpriseComponent create()
    {
        return new JornalistaMult_impl_tie();
    }


    // ====================================================================
    //
    // Métodos para implantação.
    //
    // ====================================================================

    // Método chamado pelo servidor de componentes
    public static org.omg.Components.HomeExecutorBase create_home_executor()
    {
        return new JornalistaMultHome_impl_tie();
    }

    protected void finalize()
        throws Throwable
    {
        super.finalize();
    }
}
```

C.7 Implementação do Componente Publicador

```
/*********************************************************************/
/***  Implementação do componente Noticia.Publicador.
/***  Esse componente possui uma faceta, "faceta_publicacao", que
/***  deve ser conectada ao receptáculo "recep_publicacao" de uma
/***  ou mais instâncias do componente Jornalista ou JornalistaMult.
/***  Possui também uma fonte de eventos, "fonte_noticia", que deve ser
/***  conectada ao consumidor de eventos "consumidor_noticia" de uma
/***  ou mais instâncias do componente Assinante.
/***  O publicador recebe notícias dos jornalistas através da faceta
/***  acima e as envia aos assinantes como um evento.
/*********************************************************************/

package Noticia;

public class Publicador_impl_tie
    extends    org.omg.CORBA.LocalObject
    implements org.omg.Components.SessionComponent,
               org.omg.Components.ExecutorLocator,
               CCM_Publicador_Executor,
               CCM_Publicacao

{
    // ================================================================
    //
    // Estado interno.
    //
    // ================================================================

    // Nome de uma instância do componente
    private String nome_;

    // Dimensões da janela de uma instância do componente
    private int x_, y_, altura_, largura_;

    // Referências à interface com usuário
    private javax.swing.JFrame frame_;
    private javax.swing.JTextField text_;
    private javax.swing.JTextArea textArea_;

    // Referência ao contexto em que o componente é executado
    private CCM_Publicador_Context contexto_;


    // ================================================================
    //
    // Construtor.
    //
    // ================================================================

    public Publicador_impl_tie()
    {
    }


    // ================================================================
    //
    // Métodos para a interface SessionComponent.
    //
    // ================================================================
```

```
    // Operação acionada na ativação do componente de sessão.
    public void set_session_context(org.omg.Components.SessionContext contexto)
        throws org.omg.Components.CCMException
    {
        contexto_ = (CCM_Publicador_Context) contexto;
    }

    // Operação chamada pelo contêiner (callback) para informar que o
componente está ativo.
    public void ccm_activate()
        throws org.omg.Components.CCMException
    {
    }

    // Operação chamada pelo contêiner (callback) para informar que o
componente está sendo desativado.
    public void ccm_passivate()
        throws org.omg.Components.CCMException
    {
    }

    // Operação chamada pelo contêiner (callback) para informar que o
componente está sendo destruído.
    public void ccm_remove()
        throws org.omg.Components.CCMException
    {
        // Destrói a interface com o usuário
        frame_.dispose();
        frame_ = null;
    }


    // ====================================================================
    //
    // Métodos para a interface ExecutorLocator.
    //
    // ====================================================================

    public org.omg.CORBA.Object obtain_executor(java.lang.String nome)
        throws org.omg.Components.CCMException
    {
        if (nome.equals("Publicador") || nome.equals("faceta_publicacao"))
        {
            return this;
        }
        else
        {
            throw new org.omg.Components.CCMException();
        }
    }

    public void release_executor(org.omg.CORBA.Object exc)
        throws org.omg.Components.CCMException
    {
    }

    // Encerra a fase de configuração do componente
    public void configuration_complete()
        throws org.omg.Components.InvalidConfiguration
    {
        // Verifica se a fase de configuração foi concluída.
        if(nome_ == null)
            throw new org.omg.Components.InvalidConfiguration();
```

```
    // Inicializa a interface com o usuário.

    // Cria a área principal.
    frame_ = new javax.swing.JFrame(nome_ + "'s Publicador GUI");
    // Dimensiona a janela.
    frame_.setBounds(x_, y_, largura_, altura_);

    // Cria uma área para apresentar as notícias recebidas.
    textArea_ = new javax.swing.JTextArea(15, 20);
    textArea_.setEditable(false);

    // Cria e apresenta a janela.
    javax.swing.JPanel panel = new javax.swing.JPanel(
                                    new java.awt.BorderLayout());
    frame_.getContentPane().add(panel);
    panel.add(new javax.swing.JScrollPane(textArea_),
              java.awt.BorderLayout.CENTER);
    frame_.show();
}

protected void finalize()
    throws Throwable
{
    super.finalize();
}


// ================================================================
//
// Métodos para o executor do componente
// (interface CCM_Publicador_Executor).
//
// ================================================================

// Métodos para o atributo "nome"
public void nome(String n)
{
    nome_ = n;
    if (frame_ != null) frame_.setTitle(nome_ + "'s Publicador GUI");
}

public String nome()
{
    return nome_;
}

// Métodos para o atributo "x"
public void x(int valor)
{
    x_ = valor;
    if (frame_ != null) frame_.setLocation(x_, y_);
}

public int x()
{
    return x_;
}

// Métodos para o atributo "y"
public void y(int valor)
{
    y_ = valor;
```

```
        if (frame_ != null) frame_.setLocation(x_, y_);
    }

    public int y()
    {
        return y_;
    }

    // Métodos para o atributo "altura"
    public void altura(int valor)
    {
        altura_ = valor;
        if (frame_ != null) frame_.setSize(largura_, altura_);
    }

    public int altura()
    {
        return altura_;
    }

    // Métodos para o atributo "largura"
    public void largura(int valor)
    {
        largura_ = valor;
        if (frame_ != null) frame_.setSize(largura_, altura_);
    }

    public int largura()
    {
        return largura_;
    }


    // ================================================================
    //
    // Métodos para a interface CCM_Publicador.
    //
    // ================================================================

    // O componente devolve uma referência ao objeto que implementa
    // a faceta "faceta_publicacao", ou seja, ele mesmo.
    public CCM_Publicacao get_faceta_publicacao()
    {
        return this;
    }


    // ================================================================
    //
    // Métodos para a interface CCM_Publicacao.
    //
    // ================================================================

    // Método a ser chamado pelo componente Jornalista.
    public void publica_noticia(String noticia)
    {
        // Mostra a notícia recebida.
        textArea_.append(noticia + "\n");
        // Envia a notícia como um evento a todos os
        // consumidores conectados.
        contexto_.push_fonte_noticia( new NoticiaEvent_impl(noticia) );
    }
}
```

C.8 Implementação do *Home* do Componente Publicador

```
/*****************************************************************/
/***  Implementação do home Noticia.PublicadorHome.
/*****************************************************************/

package Noticia;

public class PublicadorHome_impl_tie
    extends    org.omg.CORBA.LocalObject
    implements CCM_PublicadorHome
{
    // ================================================================
    //
    // Construtor.
    //
    // ================================================================

    public PublicadorHome_impl_tie()
    {
    }


    // ================================================================
    //
    // Métodos para a interface CCM_PublicadorHome
    //
    // ================================================================

    // Cria uma instância do executor do componente.
    public org.omg.Components.EnterpriseComponent create()
    {
        return new Publicador_impl_tie();
    }


    // ================================================================
    //
    // Métodos para implantação.
    //
    // ================================================================

    // Método chamado pelo servidor de componentes
    public static org.omg.Components.HomeExecutorBase create_home_executor()
    {
        return new PublicadorHome_impl_tie();
    }

    protected void finalize()
        throws Throwable
    {
        super.finalize();
    }
}
```

C.9 Implementação do Componente Assinante

```
/*********************************************************************/
/***  Implementação do componente Noticia.Assinante.
/***  Esse componente possui um consumidor de eventos,
/***  "consumidor_noticia", que deve ser conectado à fonte de
/***  eventos "fonte_noticia" do componente Publicador.
/***  O assinante recebe as notícias vindas do componente Publicador.
/*********************************************************************/

package Noticia;

public class Assinante_impl_tie
    extends    org.omg.CORBA.LocalObject
    implements org.omg.Components.SessionComponent,
               org.omg.Components.ExecutorLocator,
               CCM_Assinante_Executor,
               CCM_NoticiaEventConsumer


{
    // ================================================================
    //
    // Estado interno.
    //
    // ================================================================

    // Nome de uma instância do componente
    private String nome_;

    // Dimensões da janela de uma instância do componente
    private int x_, y_, altura_, largura_;

    // Referências à interface com usuário
    private javax.swing.JFrame frame_;
    private javax.swing.JTextArea textArea_;


    // ================================================================
    //
    // Construtor.
    //
    // ================================================================

    public
    Assinante_impl_tie()
    {
    }


    // ================================================================
    //
    // Métodos para a interface SessionComponent.
    //
    // ================================================================

    // Operação acionada na ativação do componente de sessão.
    public void set_session_context(org.omg.Components.SessionContext context)
        throws org.omg.Components.CCMException
    {
    }
```

```
    // Operação chamada pelo contêiner (callback) para informar que o
componente está ativo.
    public void ccm_activate()
        throws org.omg.Components.CCMException
    {
    }

    // Operação chamada pelo contêiner (callback) para informar que o
componente está sendo desativado.
    public void ccm_passivate()
        throws org.omg.Components.CCMException
    {
    }

    // Operação chamada pelo contêiner (callback) para informar que o
componente está sendo destruído.
    public void ccm_remove()
        throws org.omg.Components.CCMException
    {
        // Destrói a interface com o usuário
        frame_.dispose();
        frame_ = null;
    }


    // ================================================================
    //
    // Métodos para a interface ExecutorLocator.
    //
    // ================================================================

    public org.omg.CORBA.Object obtain_executor(java.lang.String nome)
        throws org.omg.Components.CCMException
    {
        if (nome.equals("Assinante") || nome.equals("consumidor_noticia"))
        {
            return this;
        }
        else
        {
            throw new org.omg.Components.CCMException();
        }
    }

    public void release_executor(org.omg.CORBA.Object exc)
        throws org.omg.Components.CCMException
    {
    }

    // Encerra a fase de configuração do componente
    public void configuration_complete()
        throws org.omg.Components.InvalidConfiguration
    {
        // Verifica se a fase de configuração foi concluída.
        if(nome_ == null)
            throw new org.omg.Components.InvalidConfiguration();

        // Inicializa a interface com o usuário.

        // Cria a área principal.
        frame_ = new javax.swing.JFrame(nome_ + "'s Assinante GUI");
        // Dimensiona a janela.
        frame_.setBounds(x_, y_, largura_, altura_);
```

```
    // Cria uma área para apresentar as notícias recebidas.
    textArea_ = new javax.swing.JTextArea(15, 20);
    textArea_.setEditable(false);

    // Cria e apresenta a janela.
    javax.swing.JPanel panel = new javax.swing.JPanel(
                                    new java.awt.BorderLayout());
    frame_.getContentPane().add(panel);
    panel.add(new javax.swing.JScrollPane(textArea_),
              java.awt.BorderLayout.CENTER);
    frame_.show();
}

protected void finalize()
    throws Throwable
{
    super.finalize();
}


// ========================================================================
//
// Métodos para o executor do componente
// (interface CCM_Assinante_Executor).
//
// ========================================================================

// Métodos para o atributo "nome"
public void nome(String n)
{
    nome_ = n;

    if (frame_ != null)
        frame_.setTitle(nome_ + "'s Assinante GUI");
}

public String nome()
{
    return nome_;
}

// Métodos para o atributo "x"
public void x(int valor)
{
    x_ = valor;
    if (frame_ != null)
        frame_.setLocation(x_, y_);
}

public int x()
{
    return x_;
}

// Métodos para o atributo "y"
public void y(int valor)
{
    y_ = valor;
    if (frame_ != null)
        frame_.setLocation(x_, y_);
}
```

```
    public int y()
    {
        return y_;
    }

    // Métodos para o atributo "altura"
    public void altura(int valor)
    {
        altura_ = valor;
        if (frame_ != null)
            frame_.setSize(largura_, altura_);
    }

    public int altura()
    {
        return altura_;
    }

    // Métodos para o atributo "largura"
    public void largura(int valor)
    {
        largura_ = valor;
        if (frame_ != null)
            frame_.setSize(largura_, altura_);
    }

    public int largura()
    {
        return largura_;
    }


    // ===================================================================
    //
    // Métodos para recebimento de eventos
    // (interface CCM_NoticiaEventConsumer).
    //
    // ===================================================================

    public void push_consumidor_noticia(Noticia.NoticiaEvent evento)
    {
        push(evento);
    }

    public void push(Noticia.NoticiaEvent evento)
    {
        // Atualiza a lista de notícias.
        textArea_.append(evento.texto + "\n");
    }
}
```

C.10 Implementação do *Home* do Componente Assinante

```
/*************************************************************************/
/***  Implementação do home Noticia.AssinanteHome.
/*************************************************************************/

package Noticia;

public class AssinanteHome_impl_tie
    extends    org.omg.CORBA.LocalObject
    implements CCM_AssinanteHome
{
    // ========================================================================
    //
    // Construtor.
    //
    // ========================================================================

    public AssinanteHome_impl_tie()
    {
    }


    // ========================================================================
    //
    // Métodos para a interface CCM_AssinanteHome
    //
    // ========================================================================

    // Cria uma instância do executor do componente.
    public org.omg.Components.EnterpriseComponent create()
    {
        return new Assinante_impl_tie();
    }


    // ========================================================================
    //
    // Métodos para implantação.
    //
    // ========================================================================

    // Método chamado pelo servidor de componentes
    public static org.omg.Components.HomeExecutorBase create_home_executor()
    {
        return new AssinanteHome_impl_tie();
    }

    protected void finalize()
        throws Throwable
    {
        super.finalize();
    }
}
```

C.11 Implementação do *Event Type* NoticiaEvent

```
/*********************************************************************/
/***  Implementação do event type Noticia.NoticiaEvent.
/*********************************************************************/

package Noticia;

public class NoticiaEvent_impl
     extends NoticiaEvent
{
    // ================================================================
    //
    // Construtores.
    //
    // ================================================================

    public NoticiaEvent_impl()
    {
        texto = "";
    }

    public NoticiaEvent_impl(String t)
    {
        texto = t;
    }
}
```

C.12 Implementação do *Factory* do *Event Type* NoticiaEvent

```
/**********************************************************************/
/***  Implementação do factory para o event type Noticia.NoticiaEvent.
/**********************************************************************/

package Noticia;

public class NoticiaEventDefaultFactory
    implements NoticiaEventValueFactory
{
    // ====================================================================
    //
    // Método para a interface NoticiaEventValueFactory
    //
    // ====================================================================

    // Encapsula um evento em um stream CORBA
    public java.io.Serializable
read_value(org.omg.CORBA_2_3.portable.InputStream in)
    {
        NoticiaEvent_impl v = new NoticiaEvent_impl();
        return in.read_value(v);
    }


    // ====================================================================
    //
    // Método para criar um evento.
    //
    // ====================================================================

    // Cria um novo evento.
    public NoticiaEvent create(String name)
    {
        NoticiaEvent_impl result = new NoticiaEvent_impl(name);
        return result;
    }
}
```

Referências

[ANT03] The Apache Software Foundation. *The Apache Ant Project Home Page - http://ant.apache.org/*

[BOX98] Don Box. *Essential COM*. Addison-Wesley, 1998

[BVD01] G. Brose, A. Vogel, K. Duddy. *Java Programming with CORBA, 3rd ed*. Wiley, 2001

[CCM02] Object Management Group. *CORBA Components, v. 3.0*. 2002. Documento disponível em *http://www.omg.org/cgi-bin/doc?formal/02-06-65*

[COM03] Microsoft Corporation. *Microsoft Component Object Model Home Page - http://www.microsoft.com/com/default.asp*

[COOP03] The Community OpenORB Project - *http://openorb.sourceforge.net*

[CORBA2] Object Management Group. *The Common Object Request Broker: Architecture and Specification, 2.6 ed*. 2001. Documento disponível em *http://www.omg.org/cgi-bin/doc?formal/01-12-01*

[CORBA3] Object Management Group. *The Common Object Request Broker: Architecture and Specification, 3.0 ed*. 2002. Documento disponível em *http://www.omg.org/cgi-bin/doc?formal/02-12-06*

[EJB03] Sun Microsystems, Inc. *Enterprise JavaBeans Technology Downloads & Specifications*. Especificações para as versões de EJB disponíveis em http://java.sun.com/products/ejb/docs.html

[EJCCM03] Computational Physics, Inc. *Enterprise Java CORBA Component Model Web Site - http://www.cpi.com/ejccm/*

[FPX01] F. Pilhofer. *The MICO CORBA Component Project Web Site - http://www.fpx.de/MicoCCM/*

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994

[GOAL01] Genie des Objets et composAnts Logiciels (GOAL) Team, Laboratoire d'Informatique Fondamentale de Lille (LIFL), Universite des Sciences et Technologies de Lille (USTL). *Open CORBA Component Model Platform*. 2000-2001

[HV99] M. Henning, S. Vinoski. *Advanced CORBA Programming with C++.* Addison-Wesley, 1999

[INS02] Object Management Group. *Interoperable Naming Service Specification. 1.2 ed.* 2002. Documento disponível em *http://www.omg.org/cgi-bin/doc?formal/02-09-02*

[IONA03] IONA Technologies. *ORBacus – http://www.iona.com/products/orbacus_home.htm*

[JAVA02] Sun Microsystems, Inc. *Linguagem Java – http://java.sun.com*

[LCS02] Object Management Group. Life Cycle *Service Specification. 1.2 ed.* 2002. Documento disponível em *http://www.omg.org/cgi-bin/doc?formal/2002-09-01*

[MH02] V. Marangozova, D. Hagimont. *Non-functional Replication Management in the CORBA Component Model.* Trabalho apresentado em *8th International IFIP/ACM Conference on Object-Oriented Information Systems*, Montpellier (France). 2002. Documento disponível em http://sardes.inrialpes.fr/papers/files/02-Marangozova-OOIS.pdf

[MONO03] Ximian, Inc. *The Mono Project Web Site - http://www.go-mono.com*

[NET03] Microsoft Corporation. *Microsoft .NET Home Page - http://www.microsoft.com/net*

[NSS02] Object Management Group. *Notification Service Specification. 1.0.1 ed.* 2002. Documento disponível em *http://www.omg.org/cgi-bin/doc?formal/2002-08-04*

[OPE95] Object Management Group. *ORB Portability Enhancement RFP.* 1995. Documento disponível em *ftp://ftp.omg.org/pub/docs/1995/95-06-26.pdf*

[PPR02] C. Pérez, T. Priol, A. Ribes. *A Parallel CORBA Component Model.* INRIA, 2002. Documento disponível em ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-4552.pdf

[Prit99] Jason Pritchard. *COM and CORBA Side by Side.* Addison-Wesley, 1999

[PSS01] Object Management Group. *Persistent State Service Specification, 3.0 ed*. 2001. Documento disponível em http://www.omg.org/cgi-bin/doc?ptc/2001-12-02

[RFP97] Object Management Group. *CORBA Component Model RFP*, 1997. Documento disponível em http://www.omg.org/docs/orbos/97-06-12.pdf

[RR99] C. Rocheleau, C. Rhodes. *CORBA Component Model Will Help Developers Quickly Design and Implement Mission Critical Distributed Systems.* 1999. Documento disponível em *http://www.omg.org/news/pr99/9_02a.html*

[Ruiz01] Diego Sevilla Ruiz. *CORBA & Component Model Web Site - http://www.ditec.um.es/~dsevilla/ccm/*

[SS02] Object Management Group. *Security Service Specification, 1.8 ed*. 2002. Documento disponível em http://www.omg.org/cgi-bin/doc?formal/2002-03-11

[Thom98] A. Thomas, Patricia Seybold Group. *Enterprise JavaBeans Technology – http://java.sun.com/products/ejb/white_paper.html*. 1998

[TRAD00] Object Management Group. *Trading Object Service Specification, 1.0 ed*. 2000. Documento disponível em *http://www.omg.org/cgi-bin/doc?formal/2000-06-27*

[TS02] Object Management Group. *Transaction Service Specification, 1.3 ed*. 2002. Documento disponível em *http://www.omg.org/cgi-bin/doc?formal/2002-08-07*

[W3C03] World Wide Web Consortium – *http://www.w3c.org*

[WLS00] N. Wang, D. Levine, D. Schmidt. *Optimizing the CORBA Component Model for High-performance and Real-time Applications*, apresentado na sessão *Work-in-progress* da *Middleware2000 Conference, ACM/IFIP*. 2000. Documento disponível em http://www.cs.wustl.edu/~schmidt/PDF/ middleware2000.pdf

[WSI03] Web Services Interoperability Organization – *http://www.ws-i.org*

[WSO00] N. Wang, D. Schmidt, C. O'Ryan. *Overview of the CORBA Component Model*, capítulo de G. Heineman e B. Councill (eds.), *Component-Based Software Engineering*. Addison-Wesley, 2000.

Documento disponível em http://www.cs.wustl.edu/~nanbor/papers/CBSE.pdf