UNIVERSIDADE CATÓLICA DE SANTOS
CURSO DE MESTRADO EM INFORMÁTICA

AVALIAÇÃO DO MODELO DE COMPONENTES CORBA

por

Clayton Felipe dos Santos Marques

Dissertação submetida à avaliação como requisito parcial para a obtenção do título de Mestre em Informática.

Prof.Dr. Manuel de Jesus Mendes
Orientador

Santos
2006

RESUMO

Modelos de componentes são evoluções naturais para as tecnologias de objetos distribuídos, o modelo de componentes CORBA (CCM ou *CORBA Component Model*) descreve um framework para a criação de aplicações CORBA baseadas em componentes e que encapsulam os detalhes de implementação de servidores CORBA. Componentes criados usando CCM são mais flexíveis do que os criados usando outros modelos de componentes como: COM+/.NET e EJB, em especial quanto a customização de seu comportamento.

O trabalho tem como objetivo geral avaliar os aspectos funcionais internos de um contêiner que suporta componentes CCM, para este fim desenvolveu-se um protótipo funcional que implementa um subconjunto da especificação do modelo de componentes CORBA sobre o qual se executou uma aplicação de exemplo. Os resultados das experiências realizadas durante o estudo, interpretação da especificação, a implementação e execução tanto do contêiner quanto da aplicação de testes são descritos e comentados.

ABSTRACT

Component models are natural evolutions for distributed object tecnologies. The Corba Component Model (CCM) describes a framework for component-based CORBA applications development that isolates the CORBA server implementation details from the developer. Components created usin`g CCM are more flexibles than other components created using other component models like: COM+/.NET and EJB, specialy regarding to behavior customization.

The goal of this work was evaluate some internal functional issues from a container that execute CCM components. To reach this goal, a functional prototype that implements a CCM specification sub-set was developed and an example application was executed using the prototype. The results obtained during the specification understanding, prototype and example implementation and execution was commented and described.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

ORB – *Object Request Broker* – é um conjunto de programas que media a troca de mesagens entre dois objetos potencialmente remotos.

POA – *Portable Object Adapter*, elemento de software cuja função é interceptar as mensagens destinadas a um objeto CORBA e encaminha-las a um programa que implementa o comportamento esperado.

COM – *Component Object Model,* sigla que identifica o modelo de obejtos distribuídos oferecido pela Microsoft.

DCOM – *Distributed COM*, sigla que identifica o padrão para objetos distribuídos da Microsoft.

DCE RPC – *Distributed Computing Environment Remote Procedure Call*, representa um padrão de execução remota de programas, mas aplicando o conceito de procedimentos (*procedures*) em lugar de objetos.

JVM – *Java Virtual Machine*. Software que permite a execução de uma aplicação compilada usando-se a linguagem de programação Java.

JNDI – *Java Naming Directory Interface*. Padrão definido pela arquitetura J2EE que define um conjunto de interfaces que dão acesso a um serviço de registro e localização de referências a objetos para servidores que respeitam a arquitetura J2EE.

JAR – *Java Archive*, Extensão que identifica um arquivo compactado usando-se o utilitário "JAR". Contém programas compilados em Java.

MTS – *Microsoft Transaction Server*. Conjunto de softwares da Microsoft que oferece o ambiente para a execução de componentes COM transacionais.

MSMQ – *Microsoft Message Queue*. Software da Microsoft que gerencia o envio e entrega de mensagens assíncronas.

ACID – Acrônimo para as palavras: "Atomicidade", "Consistência", "Isolamento" e "Durabilidade" que são as características que devem existir em uma transação.

DLL – *Dynamic Linked Library* são bibliotecas que contém softwares que respeitam o padrão COM.

IDE – *Integrated Development Environment*, identifica um software usado para auxiliary o processo de codificação em uma determinada linguagem.

JDK – *Java Developer Kit*, identifica o conjunto de programas e bibliotecas (arquivos JAR) que devem estar disponíveis em um computador para desenvolver aplicações em Java.

WSDL – *Web Service Description Language*, nome da linguagem padrão para a declaração de interfaces para serviços Web.

SOAP – *Service Oriented Application Protocol*, nome de um protocolo que opera em nível de aplicação (OSI), seu papel na arquitetura SOA é de transportar as requisições do clinete ao servidor usando.

SUMÁRIO

1 INTRODUÇÃO

Tecnologias para objetos distribuídos encapsulam os pormenores ligados à comunicação de dados e ao empacotamento e desempacotamento (*marshaling* e *unmarshaling*) dos dados trocados entre os elementos de software que compõem uma aplicação. Estes elementos são projetados e implementados como objetos que podem potencialmente ser acessados remotamente. Objetos distribuídos oferecem um grau de abstração mais alto em relação aos sistemas construídos sobre *sockets* ou mesmo sobre RPC [SER99].

Porém, ainda assim, exige-se investimento de tempo: no projeto, na seleção e na implementação de estratégias para o uso de serviços como: localização de objetos (serviço de nomes), controle de filas de mensagens assíncronas (serviço de notificações ou eventos), entre outros.

Um modelo de componentes é uma opção de arquitetura que oferece um *framework* para, indiretamente, dar acesso aos elementos de software construídos sobre uma tecnologia de objetos distribuídos, mas permitindo projetar os elementos que compõem o sistema em unidades mais abstratas, os componentes.

No projeto de software baseado em componentes, o elemento chave continua sendo a interface, mas as estruturas que compõem o sistema são elementos mais abstratos que os objetos, os componentes. Componentes oferecem portas de comunicação para que se possa definir as associações possíveis entre eles (conexões).

Ao declarar as interfaces dos componentes que formam uma aplicação ou as conexões existentes entre eles, o desenvolvedor não tem que se preocupar com os objetos que implementam cada interface ou representam cada uma das portas e conexões.

Objetos e componentes podem ser projetados independentemente da tecnologia que será usada para implementa-los, a principal diferença entre as duas estruturas é que a segunda pode ser identificada e projetada mais cedo no projeto (componentes se encontram mais distantes da implementação que os objetos). A transição do componente de uma unidade de composição para várias unidades de execução acontece durante o processo de *deployment*.

Contudo, os principais modelos de componentes são construídos sobre alguma tecnologia para objetos distribuídos de forma que, mesmo sem o conhecimento do desenvolvedor, os objetos são criados e executados através de adaptadores [GAM00] (programas que interceptam as requisições recebidas para a "interface do componente" e as convertem em requisições a objetos que internamente formam o componente).

Assim, pode-se inferir que quanto mais versátil a tecnologia de objetos distribuídos, mais flexibilidade e recursos terão os modelos de componentes construídos sobre ela.

Partindo-se desta dedução, CORBA [OMG02b] é uma escolha natural para suportar um modelo de componentes, dado que [SIE00] [BOL02]:

- É uma tecnologia estável e madura;
- É implementada em produtos de diversos fabricantes;

- Aplicações desenvolvidas para CORBA podem ser executadas em várias plataformas;
- Integra-se com as demais tecnologias de objetos distribuídos;
- Aplicações desenvolvidas para CORBA podem ser escritas em várias linguagens;
- Oferece um efetivo controle no ciclo de vida dos objetos o que permite ajustar a sua forma de execução (comportamento) conforme a necessidade;
- Permite instalação distribuída.

O modelo de componentes CORBA (CCM – *Corba Component Model*) é um modelo de componentes construído sobre CORBA e incorporado à especificação CORBA 3.0 [OMG02b]. Como CORBA é um padrão especificado pelo OMG (*Object Management Group*), conseqüentemente, o modelo de componentes CCM [OMG02] é compatível com os outros padrões definidos pela instituição.

CCM é um modelo de componentes altamente adaptável. Um servidor CCM pode ser desenvolvido para suportar vários ORBs [OMG02b] e conseqüentemente diversas linguagens de programação e plataformas de execução.

Dentre os modelos de componentes existentes, o modelo CCM é o mais recente e aplica os conceitos mais modernos sobre computação baseada em componentes [SZY03] como: configuração, portas e conexões, um conjunto de predicados que poderia classificá-lo como uma ótima e, em alguns casos, como a melhor opção entre os modelos de componentes. Mas a adoção de CCM como um padrão do mercado ainda não aconteceu.

Diferentemente de CORBA, o modelo de componentes CORBA é uma tecnologia não amadurecida e deve ser testada. Embora a especificação final tenha sido publicada em 2002 [OMG02], existem poucas implementações que usam o modelo CCM.

Em janeiro de 2002 havia 12 projetos em andamento para a construção de produtos que implementavam o modelo CCM [OMG02c] dos quais, ao final de 2005, somente 6 projetos continuam em andamento, os demais foram descontinuados. Além destes projetos, existe uma implementação completa, comercial que foi desenvolvido posteriormente [ICM06].

Por outro lado, o modelo de componentes CORBA incorpora as melhores características existentes nos demais modelos como: vários tipos de componentes, cada um deles com comportamentos diferentes, como no modelo EJB [SUNb][ROM02], qualquer um dos tipos de componentes pode executar operações assíncronas em outros componentes e oferecer várias interfaces, como no caso dos componentes MTS [ROF99].

Este trabalho se propõe a analisar o funcionamento interno do modelo de componentes CORBA sob a ótica do desenvolvedor de um ambiente de execução para componentes (contêiner), dado que o texto da especificação é bastante complexo e de difícil interpretação, o trabalho se porpõe a oferecer uma interpretação da especificação que deve permitir compreender efetivamente a arquitetura do modelo de componentes. Para este fim, foi desenvolvido um protótipo funcional que implementa um subconjunto da especificação CCM.

A especificação CCM descreve quatro tipos de portas [OMG02]: facetas, receptáculos, emissores de eventos e receptores de eventos. Usando-se o protótipo, pretende-se identificar como a arquitetura CORBA é encapsulada para que vários componentes mantenham conexões entre as diversas portas e como faz uso do serviço de notificações CORBA [OMG02a], sem que o desenvolvedor do componente se preocupe com estes pormenores.

Para o modelo CCM, existem 5 categorias de componentes: serviço, sessão, processo, entidade e vazio. O protótipo foi construído para atender às características de componentes de "sessão" devido ao fato de terem que manter estado e o registro das conexões entre as portas permitindo observar a influência das políticas POA [VIN98] [PIL99] sobre os objetos CORBA que formam os componentes.

Enfim, os objetivos específicos deste trabalho são:

- Descrever e comparar as principais tecnologias de objetos distribuídos;
- Descrever e comparar as características dos principais modelos de componentes;
- Compreender o funcionamento interno de um contêiner para componentes do tipo CCM.
- Avaliar o protótipo de contêiner para a execução de componentes CCM estendidos do tipo *sessão* que implementa um sub-conjunto da especificação [OMG02] para auxiliar a sua interpretação.
- Avaliar uma aplicação construída usando-se o modelo CCM e executá-la no protótipo.

Para alcançar estes objetivos, o trabalho foi organizado de forma a inicialmente, no capítulo 2, descrever e comparar os principais modelos usados para a criação de objetos distribuídos: CORBA, RMI [FAR98][HOR01] e COM/DCOM [SER99].

Em seguida, ainda no capítulo 2, define-se os conceitos de componentes de software e faz-se a descrição e posterior comparação entre os principais modelos de componentes: EJB [SUNb][ROM02], CCM e COM+/MTS [KIR97][KIR97a].

No capítulo 3 a especificação do modelo de componentes CCM é descrita para oferecer o entendimento necessário para que o leitor possa entender os aspectos discutidos no capítulo seguinte, que explora a arquitetura do protótipo e como ele foi projetado para atender à especificação.

Para testar o funcionamento do protótipo e avaliar se ele realmente atendeu às características esperadas, foi implementada uma versão do problema dos "Filósofos glutões".

A modelagem do problema na forma de componentes CCM é descrita no capítulo 5 assim como os aspectos ligados à conversão de interfaces e ao funcionamento da aplicação no contêiner e sua instalação.

2 CONCEITOS SOBRE OBJETOS DISTRIBUÍDOS E MODELOS DE COMPONENTES

2.1 OBJETOS DISTRIBUÍDOS

Alguns sistemas precisam que seus módulos possam estar localizados em computadores diferentes e, para tornar possível a interação entre estes elementos de software distribuídos, pode-se aplicar diversas técnicas de programação, entre elas, talvez a mais simples, o uso de sockets (sob a arquitetura cliente-servidor)[SER99].

O problema com esta abordagem é que todo o controle da comunicação entre o cliente (parte do software que solicita a execução de uma funcionalidade no módulo remoto) e o servidor (software que atende a requisição do cliente) é responsabilidade do programador.

É desejável isolar da implementação do software os detalhes envolvidos com a comunicação de dados propriamente dita, como: endereçamento dos nós de rede remotos (localização dos elementos da aplicação), forma de empacotamento de dados (composição das mensagens para envio de parâmetros e obtenção de resultados), garantia da confiabilidade dos dados recebidos, etc...

Softwares que oferecem estas características (que isolam da implementação dos elementos de software os detalhes ligados à comunicação de dados) são chamados de *middleware* [SER99].

Quando uma aplicação distribuída faz uso dos conceitos de orientação a objetos, diz-se que usa a tecnologia de "computação por objetos distribuídos" (DOC – *Distributed Object Computing*). Neste caso, o *middleware* usado pela aplicação acessa seus elementos distribuídos como objetos e não como procedimentos (*procedures*) ou funções (*functions*) [SER99].

As principais tecnologias para objetos distribuídos (ou *middlewares*) são: CORBA, RMI e DCOM [CIC99] [MAR00].

2.1.1 CORBA

CORBA (*Common Object Request Broker Architecture*), é uma descrição de arquitetura para a criação e execução de objetos distribuídos especificada pelo OMG [OMG02b][BOL02][SIE00][ORF98].

Aplicações que usam CORBA são softwares orientados a objetos onde o elemento chave de seu projeto é a interface e não a classe. Para que objetos possam ser acionados remotamente, eles devem ter suas interfaces declaradas usando-se a linguagem IDL (*interface definition language*). Para CORBA, a invocação de operações pode ser feita de forma estática ou dinâmica.

Na invocação estática, o que torna possível a execução de operações em um objeto remoto e a obtenção de sua resposta é a existência de *proxies* [GAM00] que interceptam as requisições localmente, as prepararam para que sejam transmitidas (*marshaling*) e efetivamente as transmitem.

Na nomenclatura CORBA, o *proxy* do lado cliente é chamado *stub* e o *proxy* do lado do objeto é chamado *skeleton*.

A criação destes *proxies* acontece durante a compilação das interfaces, um processo realizado por um software "compilador de IDL" que realiza a leitura dos arquivos que contém as definições das interfaces (arquivos IDL) e gera os *proxies*. O processo de geração de *proxies* é representado na figura 2-1:

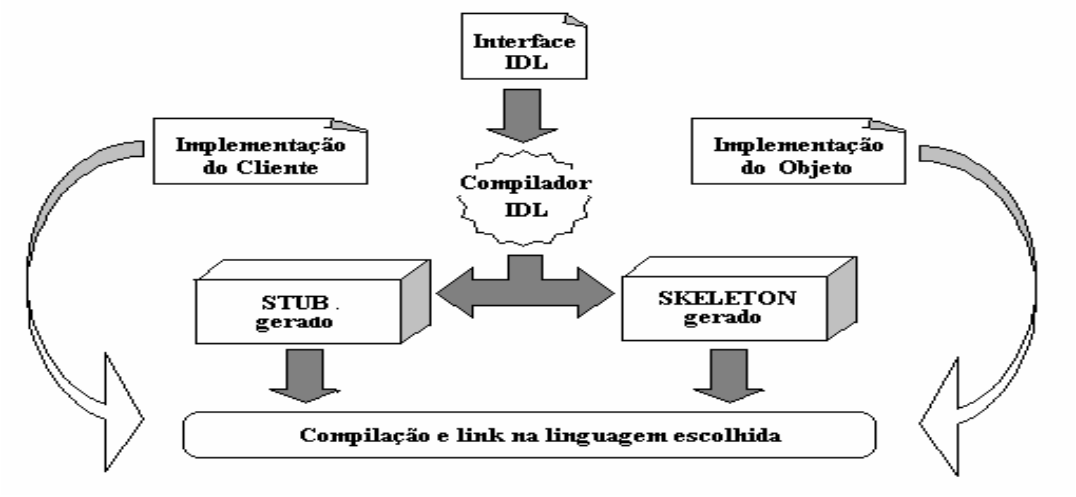

Figura 2-1- A compilação das interfaces escritas em IDL gera os *proxies*.

Quando um cliente solicita a execução de uma operação em um objeto remoto, ele a faz no *stub* (que representa o objeto remoto localmente) o que permite que a implementação de um objeto remoto seja totalmente independente do cliente, desta forma, o objeto remoto pode ser implementado em uma linguagem diferente ou ser executado em um ambiente diferente do cliente e dos demais objetos que compõem uma aplicação.

No computador onde o objeto CORBA é hospedado (chamado de agora em diante de *servidor*), existe a figura do *Servant*.

O *Servant* é o software (ou conjunto de softwares, dependendo da linguagem em que for codificado) que contém a implementação das operações descritas nas interfaces IDL. Os *Servants* são executados no servidor, quando uma operação que ele implementa é requisitada por um cliente.

A seleção, acionamento e controle do ciclo de vida (instanciação, ativação, desativação e destruição) dos se*rvants* é responsabilidade do POA (*Portable Object Adapter*) [PIL99].

O POA, um objeto que implementa o padrão de projetos adaptador (*adapter*) [GAM00], intercepta as requisições recebidas pelo servidor e destinadas aos objetos CORBA gerenciados por ele e escolhe qual das instâncias de *servant* disponíveis vai executar a requisição (ou se será necessário obter uma nova instância).

O cliente acessa os objetos por meio de uma referência ao objeto (*object reference*), uma estrutura formada por [ORF98] [SIE00]:

- Um endereço que permite ao *stub* localizar o computador onde o objeto CORBA é executado.
- A identificação do adaptador de objetos que criou a referência ao objeto
- Uma identificação de objeto

Para obter a referência ao objeto, o cliente deve solicitar ao ORB a referência a algum serviço que permita a sua obtenção, por exemplo, o serviço de nomes (*Naming Service*).

A figura 2-2 representa a interação entre: o cliente, o *stub*, o *skeleton*, o POA (representado na figura como ”adaptador de objetos”), o *Servant* (representado na figura como “implementação do objeto”) e o ORB (figura baseada em [OMG02b]):

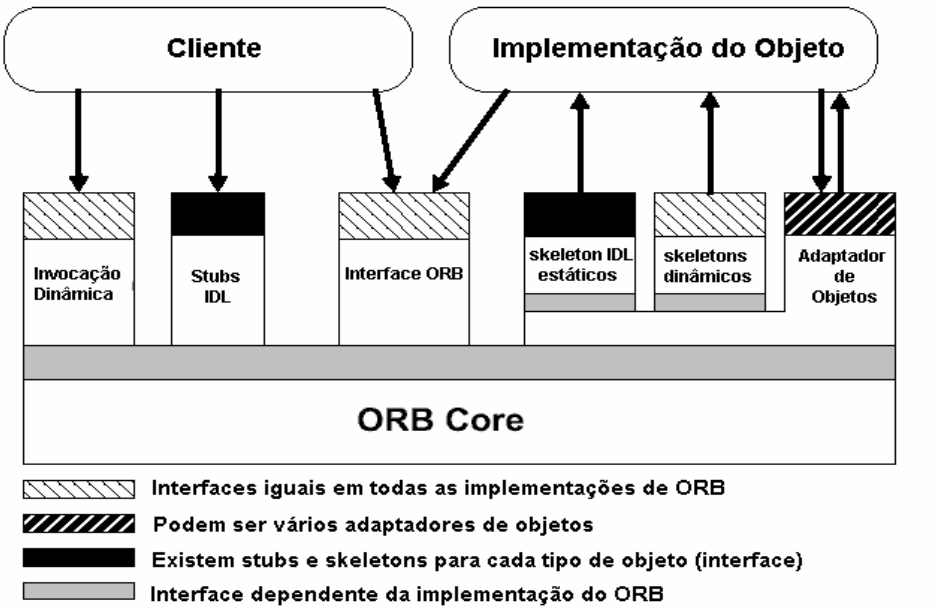

Figura 2-2 - Interação entre um cliente e um objeto CORBA durante uma invocação de operação

O elemento central da arquitetura de uma aplicação CORBA é o ORB (*Object Request Broker*) que, em uma tradução livre, pode ser entendido como: mediador para requisições a objetos.

CORBA descreve quais interfaces devem ser oferecidas por um ORB e qual o comportamento esperado para cada operação nelas descritas.

O ORB é o conjunto de programas que participa da comunicação para o acionamento de operações em objetos potencialmente remotos (diz-se potencialmente porquê nada impede que os objetos estejam em execução no mesmo computador que os clientes). Estes programas são desenvolvidos por um fabricante e implementam as interfaces e comportamentos descritos na especificação CORBA.

Observando-se a figura 2-2, com exceção do cliente e do *servant* (implementação do objeto), todos os elementos representados formam o ORB.

2.1.2 RMI

RMI (*Remote Method Invocation*) é um *middleware* que permite a criação de aplicações que usam objetos distribuídos, mas somente para objetos escritos usando-se a linguagem Java [HOR01][SER99][FAR98].

A declaração de um objeto RMI (objeto que pode ser acessado remotamente) começa pelas interfaces que ele implementa, estas interfaces devem ser do tipo "Remoto" previsto na API da linguagem Java (*java.rmi.Remote*) e podem ser criadas manualmente ou geradas a partir de arquivos com declarações escritas em IDL (padrão CORBA). Este tipo de interface será referenciado neste texto como "interface remota".

A classe que declara um objeto RMI deve implementar uma ou mais interfaces remotas e herdar a classe *java.rmi.server.RemoteServer* ou uma de suas sub-classes.

Para RMI também existe a figura dos *proxies*, que usam a mesma nomenclatura CORBA onde o *proxy* usado no lado cliente é chamado de *stub* e o *proxy* no lado do objeto RMI é chamado de *skeleton*.

Os *proxies* são gerados por um utilitário "compilador de RMI" chamado "rmic" que é oferecido nas distribuições da linguagem Java. Mas, diferente do que acontece em CORBA, a compilação não é feita sobre a interface, mas sobre a classe que implementa a interface remota na forma de *bytecode* (formato que uma classe assume depois de ter sido compilada pelo compilador da linguagem Java).

A descrição realizada acima está representada na figura 2-3:

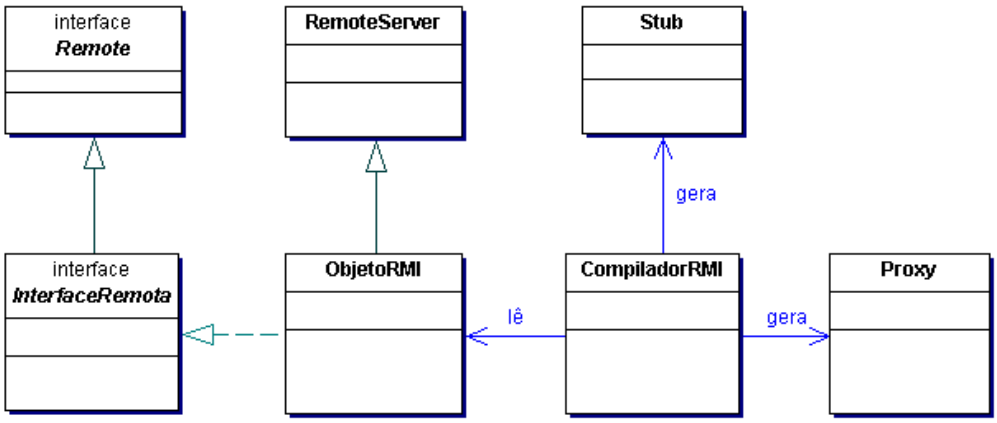

Figura 2-3 - Arquitetura de um objeto RMI

Para serem localizados e terem suas operações acionadas pelos clientes, os objetos RMI devem ser registrados no "*RMI registry*" uma entidade que, se comparado com CORBA, acumula funções: do ORB, do adaptador de objetos, do repositório de interfaces e do serviço de nomes. É um processo que deve ser executado no equipamento que hospeda os objetos remotos.

2.1.3 DCOM

De forma similar aos demais modelos descritos, no modelo DCOM o acesso às funcionalidades de um objeto é realizado a partir das operações definidas em suas interfaces. As interfaces dos objetos DCOM são declaradas usando-se a linguagem Microsoft IDL (MIDL – *Microsoft Interface Definition Language*), que incorpora características de orientação a objetos na IDL usada no padrão DCE RPC. A compilação do arquivo MIDL gera o *proxy* e o *stub* (*proxies* análogos respectivamente a *stubs* e *skeletons* no modelo CORBA).

Um objeto DCOM implementa a interface *IUnknown* da API COM e suas operações são acessáveis remotamente (ou localmente, a implementação do cliente não é alterada em relação ao modelo COM, o que se altera são parâmetros incluídos ou ajustados no registro do sistema operacional Windows – *windows registry* - garantindo para o cliente a independência quanto à localização do servidor onde o objeto DCOM pode ser acionado).

Além de implementar a interface *IUnknown*, o objeto COM/DCOM deve implementar também as interfaces contendo as operações particulares daquele objeto, como representado na figura 2-4 (baseada em [SER99]):

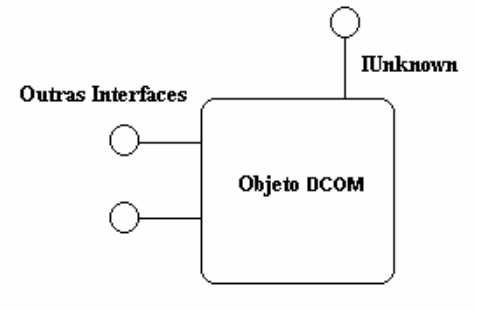

Figura 2-4 - Interfaces de objetos DCOM

As interfaces de um objeto DCOM podem ser acessadas através dos "*interface pointers"*. A obtenção de um "*interface pointer"* acontece quando o cliente solicita a instanciação do objeto COM/DCOM, atividade executada por meio de uma operação (disponível na API COM/DCOM) chamada `CoCreateInstance` que recebe como parâmetro o CLSID do objeto COM/DCOM.

O CLSID (*class identifiers*) é a identificação de um objeto DCOM, trata-se de um GUID (*globally unique identifiers,* um identificador de 128 bits usado para registrar elementos de software no registro disponível no sistema operacional windows - *windows registry*) que é usado para identificar classes.

O SCM (*service control manager* ou gerenciador de controle de serviços) é o serviço responsável pela ativação de objetos por demanda. No lado servidor o SCM, por meio das bibliotecas DCOM, localiza no registro do sistema operacional (*registry*) qual DLL ou executável detém a classe para instanciar o objeto pedido, instancia o objeto, cria um *interface pointer* e o envia de volta para o cliente.

Quando um cliente aciona alguma operação da API DCOM que solicita a obtenção do *interface pointer* de um determinado objeto, o SCM do lado cliente aciona operações no SCM do computador servidor solicitando a instanciação do objeto identificado pelo cliente.

Uma vez que o cliente obteve o *interface pointer* para o objeto identificado pelo CLSID, este já pode iniciar a execução das operações no objeto remoto.

A figura 2-5 representa um resumo da arquitetura DCOM, onde os elementos gerados pela compilação dos arquivos MIDL (*proxy* e *stubs*) são incorporados e como participam da requisição e obtenção da resposta entre cliente e objeto DCOM.

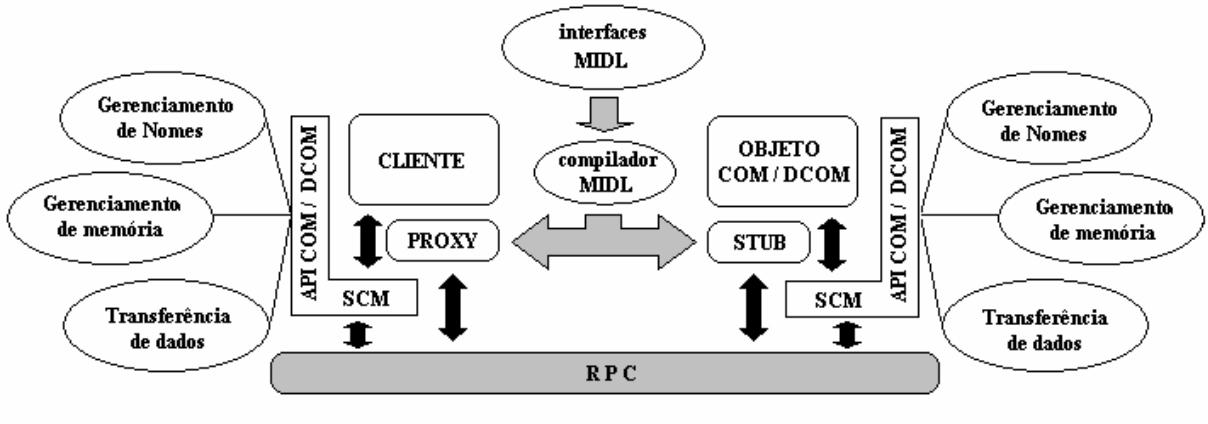

Figura 2-5 – Arquitetura do modelo de COM/DCOM

2.2 COMPARAÇÃO ENTRE MODELOS DE OBJETOS DISTRIBUÍDOS

As principais tecnologias para objetos distribuídos definem:

- Uma linguagem para declarar interfaces;
- Um processo de compilação de interfaces para geração de *proxies*;
- Um mecanismo de localização de objetos e acionamento das operações nos objetos do servidor;
- Uma forma padronizada de obter as referências remotas aos objetos (a partir do cliente) para que possam identificá-los junto ao servidor.
- Uma API para a implementação dos objetos e clientes.

Mas diferem sob vários pontos de vista, como se representa na tabela 2.1.

Tabela 2-1 - Comparação entre modelos para a confecção de objetos distribuídos

CORBA	DCOM	RMI
DECLARAÇÂO DE INTERFACES		
Utiliza a linguagem CORBA IDL (*Interface Definition Language*)	Utiliza a linguagem Microsoft IDL (*Interface Definition Language da Microsoft*)	Interfaces são implementadas na própria linguagem Java
INTERFACES PADRÃO		
Todas as interfaces herdam as características de *CORBA::Object*	Todos os objetos implementam a interface *IUnknown*	Todos os objetos servidores implementam *java.rmi.Remote*

ADAPTADOR DE OBJETOS		
Localização de objetos e encaminhamento de requisições de clientes é responsabilidade do adaptador de objetos .	A localização e acionamento de operações nos objetos são responsabilidade do serviço SCM (*service control manager*).	A JVM é responsável pelo carregamento de objetos em memória e execução de suas operações.
COMO IDENTIFICAM OBJETOS		
Identifica objetos por meio de "referências a objetos" (*object references*). Estas referências são obtidas por meio de serviços de localização de objetos como: serviço de nomes (*naming service*).	Identifica um objeto remoto por meio de um "*interface pointer"* que são obtidos por meio de: *Monikers*, identificação manual da localização dos servidores ou por meio de consultas ao *registry*.	Identifica objetos por meio de uma referência ao objeto obtida no registro RMI.

Outros aspectos não são semelhantes, dependem intimamente das opções de projeto dos modelos, estes aspectos estão agrupados na Tabela 2.2:

Tabela 2-2 - Comparação entre modelos para a confecção de objetos distribuídos

CORBA	DCOM	RMI
PORTABILIDADE		
O ORB é o elemento central da arquitetura CORBA. Para que este seja executado em outra plataforma, basta gerá-lo novamente nesta nova plataforma (compilação de código fonte e IDL) esta operação pode ser realizada no ORB que hospedará o objeto na Nova plataforma.	O modelo DCOM é muito ligado ao sistema operacional da Microsoft (o conjunto de serviços usado pelo modelo é oferecido pelo sistema operacional). Mas pode ser executado em qualquer plataforma onde os serviços COM possam ser instalados.	RMI herda a portabilidade nativa da linguagem Java. Objetos RMI podem ser instalados em qualquer plataforma sem nenhum ajuste, basta para isso que haja uma implementação da JVM disponível na plataforma desejada.
LINGUAGEM DE PROGRAMAÇÂO		
Podem ser criados objetos em qualquer linguagem que tenha APIs para CORBA	Podem ser criados objetos em qualquer linguagem que tenha API compatível com o modelo COM.	RMI é intimamente ligado à linguagem Java, mas, pode executar objetos escritos em qualquer linguagem, contanto que estejam sob o formato *bytecode*.

Para uma aplicação construída sobre um modelo de objetos distribuídos a interface é o elemento chave. É a partir das interfaces que os *proxies* são criados e usados localmente para possibilitar o acionamento de operações em objetos remotos de forma transparente.

Mas o desenvolvimento de um objeto distribuído exige que o desenvolvedor do objeto codifique atividades como: obtenção de referência a um serviços de nomes, publicação do objeto no serviço de nomes, registro de consumidores e fornecedores de mensagens para usar um serviço de filas ou de distribuição de mensagens assíncronas (serviço de eventos ou notificações), acionamento e finalização do controle de transações ou garantir autenticação e verificar se um determinado aplicativo pode acionar operações do objeto por meio de um serviço de segurança.

Este fato leva o desenvolvedor a investir tempo de projeto e de codificação nestes detalhes de implementação. Além disso, desenvolvedores participantes de equipes

diferentes podem optar por estratégias diferentes de acionamento a serviços o que pode dificultar ou mesmo impossibilitar a reutilização do objeto em aplicações diferentes.

Uma opção para superar as limitações existentes nos modelos de objetos distribuídos, é padronizar a forma como os serviços são acessados, aumentando o nível de abstração dos elementos de projeto fazendo com que o desenvolvedor passe a se preocupar exclusivamente com as interfaces oferecidas e com as conexões existentes entre estas interfaces e esqueça as questões ligadas com o acesso aos serviços, criando assim, unidades mais independentes com foco no projeto e não na execução.

2.3 COMPONENTES E MODELOS DE COMPONENTES

Componentes são elementos de software distribuídos ou não, potencialmente reusáveis que são acessados por meio de suas interfaces e são construídos respeitando a um modelo de componentes. Mas, principalmente, componentes são unidades de instalação e composição que são executadas em contêineres e podem ser obtidas de terceiros [SZY02][SZY03].

Assim como acontece aos objetos desenvolvidos sobre as tecnologias para objetos distribuídos, componentes podem ser executados localmente ou ser acessados de um computador remoto.

A principal característica de um componente é o fato de ser potencialmente reusado em outras aplicações, bastando para isso que cada um dos clientes que façam uso do componente estejam preparados para acessar suas interfaces. Vale lembrar que estas características (poder ser usado novamente em outro sistema sem a necessidade de re-compilação e ter a implementação do software independente da interface declarada para acessá-la) também existem para objetos distribuídos.

De forma similar ao que acontece aos objetos quando se usam tecnologias para objetos distribuídos, componentes são construídos sobre um modelo de componentes.

Um modelo de componentes é um conjunto de normas e definições que especificam como os componentes devem ser escritos e como um contêiner deve ser construído para hospedar estes componentes.

A entidade central de um modelo de componentes é o contêiner. Trata-se de um conjunto de elementos de software que gerencia o ciclo de vida dos componentes. O contêiner fornece aos componentes, acesso aos serviços por ele oferecidos (o conjunto de serviços oferecidos por um contêiner e a forma como estes serviços podem ser obtidos por um componente também é especificado no modelo de componentes).

Contêineres são mediadores entre: o componente, os recursos disponíveis no ambiente de execução e o cliente. Para que um cliente possa acessar as operações de um componente, ele precisa obter a referência ao componente, esta referência é obtida através do contêiner.

As interfaces acessadas pelos clientes são oferecidas pelo contêiner que, ao receber uma requisição a alguma de suas operações, a intercepta e direciona para o elemento de software que internamente é capaz de atender àquela requisição, atuando como um adaptador (padrão de projetos *adapter*).

Para que os elementos de software que formam internamente um contêiner possam entrar em execução para atender a um determinado componente, este deve passar por um processo de instalação.

Neste trabalho, a palavra "instalação" é usada como tradução para *deployment* que é um processo automático de preparação de cada componente para a execução em um determinado ambiente.

O processo de instalação ou *deployment* faz a leitura de um arquivo que contém a descrição do componente (seu nome, quais são os elementos que o formam, como as operações declaradas em suas interfaces externas devem ser mapeadas para os elementos de software que as implementam internamente, quais serviços devem ser usados e quando estes devem ser acionados, entre outras informações – dependendo do modelo de componentes).

A partir da leitura deste arquivo (chamado de descritor de instalação), o software que realiza a instalação, cria código de apoio (novos programas) ou re-compila os programas já oferecidos para que o componente possa ser acionado no novo ambiente de execução.

Componentes existem no mesmo nível dos pacotes, módulos ou classes e não no nível dos objetos ou objetos distribuídos [SZY02] isso ocorre porque componentes são declarações e não unidades de execução (não é possível executar um componente assim como não é possível executar uma classe ou um pacote).

Os componentes são formados internamente por estruturas de software menores em um nível de abstração mais baixo (objetos, procedimentos ou funções) que além de oferecerem a execução propriamente dita das operações expostas pelas interfaces do componente oferecem também meios de interação entre o componente e o container.

Quando se cria efetivamente um componente, não se escreve somente o código, mas também arquivos contendo configurações que descrevem como deve funcionar em cada ambiente (como interagem com o conteiner), enfim, componentes são unidades independentes quanto a produção, aquisição e instalação, que interagem para formar o funcionamento de um sistema [SZY02].

Graças ao processo de instalação, uma vez que um desenvolvedor já declarou as interfaces do componente, forneceu o código que executa as operações e os descritores de instalação, todos estes elementos são empacotados em um arquivo (normalmente um arquivo compactado) para que possa ser levado a outros ambientes de execução.

Os principais modelos de componentes existentes são:

- **EJB** – *Enterprise Java Beans*, modelo de componentes definido pela *Sun Microsystems*. O *framework* usado para a confecção dos componentes é especificado e fornecido pela *Sun* que centraliza o desenvolvimento e evolução do modelo de componentes. Os componentes criados usando o padrão EJB são instalados em servidores de aplicação de diversos fabricantes (que respeitam a especificação de arquitetura J2EE e que implementam o modelo de componentes EJB) e utiliza como *middleware*: RMI, CORBA, ou soluções *ad-hoc*, dependendo da opção do fabricante que implementou o servidor de aplicações [SUNb][SUN02][ROM02] .

- **CCM** – Modelo de componentes CORBA (*Corba Component Model*), modelo de componentes proposto pelo OMG (*Object Management Group*) como parte da especificação CORBA 3.0. Os componentes criados usando o padrão CCM podem ser instalados em servidores de diversos fornecedores (que respeitam a especificação CCM) e utiliza CORBA como *middleware*. [OMG02][BOL02]
- **COM+** – Modelo de componentes transacionais da Microsoft, criado baseado no modelo de componentes COM. Os componentes criados usando o padrão COM+ são instalados em servidores MTS (*Microsoft Transaction Server*). Utiliza DCOM como *middleware* [MEY99].

2.3.1 EJB

Um componente EJB é uma estrutura de programação formada por um conjunto de classes escritas em linguagem Java que podem ser acessadas por meio de interfaces: uma para o controle do ciclo de vida do componente (solicitação de uma nova instância do componente para uso do cliente ou solicitação da liberação do componente para o contêiner) e outra para o acionamento das operações de negócio do componente.

O modelo de componentes EJB prevê três (3) grupos de componentes:

- **EJB de Sessão (*Session Beans)***: São componentes "transientes", isto é, que não mantém seu estado (valores das variáveis de instância do objeto que efetivamente executa as operações da interface oferecida pelo componente) após o término da execução do cliente que requisitou a sua criação. São componentes usados para implementar lógica de negócio propriamente dita, suas operações devem representar a execução de operações de negócio. Existem duas variações deste tipo de componente:
  - **Sem Estado (*Stateless)***: Que não mantém o seu estado após cada chamada de operação. Seu tempo de vida é limitado ao da execução da operação solicitada pelo cliente. Suas operações implementam atividades como: consultas a tabelas (ou sistemas legados), execução de cálculos, etc...
  - **Com estado *(Stateful[1])***: Que mantém o seu estado mesmo depois de finalizado o acionamento da operação. O tempo de vida deste tipo de componente é limitado ao tempo de vida da sessão de conexão do cliente (enquanto o cliente que o acionou estiver com uma sessão ativa no servidor, o componente manterá o seu estado). O componente é destruído após uma solicitação do cliente ou pelo contêiner, se a sessão do cliente que solicitou o componente não for mais válida.
- **EJB de Entidade (*Entity Bean)***: São componentes "persistentes" concebidos para que, cada uma de suas instâncias represente uma relação (linhas compostas de uma ou mais tabelas).

[1] A especificação do modelo de componentes EJB [SUNb] nomeia os componentes de sessão "com estado" como "s*tateful*" grafado com somente um "l" e não "*statefull*" como o nome leva a deduzir.

Estes componentes não foram concebidos para serem usados para acionamento de operações de negócio. Suas operações são relacionadas à obtenção ou alteração de dados ou "CRUD": ***C**reate* (inclusão), ***R**ecall* (consulta), ***U**pdate* (atualização de dados), ***D**elete* (exclusão). EJBs de Entidade têm sua persistência gerenciada de duas formas:

- **CMP**: *Contêiner Managed Persistence* (persistência gerenciada pelo contêiner). Neste caso, o sincronismo com a fonte de dados é responsabilidade do contêiner. O implementador não realiza inserções no código para realizar atualizações ou consultas nas fontes de dados. Fica sob a responsabilidade do elemento que construir os descritores de instalação, introduzir as consultas manualmente.
- **BMP**: *Bean Managed Persistence* (persistência gerenciada pelo componente). Neste caso o implementador do componente deve codificar todos os passos envolvidos na consulta e atualização dos dados nas fontes de dados do sistema.

- **EJB orientados a mensagens (*Message-driven EJB*)**: Similares aos EJB de sessão "sem estado", com a diferença que as suas operações são acionadas por meio de mensagens assíncronas.

Um componente EJB é formado por um conjunto de classes escritas em linguagem Java que são implementadas pelo desenvolvedor do software ou geradas pelo contêiner. São elas:

- **Executor do componente (*Enterprise Bean*)**: É a classe que contém a implementação propriamente dita das operações do componente. Do ponto de vista da linguagem, uma classe deste tipo deve implementar uma interface derivada de *javax.ejb.EnterpriseBean* dependendo do tipo de componente sendo criado:
  - EJBs de Sessão implementam: *javax.ejb.SessionBean*
  - EJBs de Entidade implementam: *javax.ejb.EntityBean*
  - EJBs orientados a Mensagens implementam: *javax.ejb.MessageDrivenBean*
- **Objeto EJB**: Toda a comunicação entre o cliente e o executor do componente é mediada pelo objeto EJB. O objeto EJB é implementado pelo próprio contêiner durante o processo de instalação com base na leitura dos métodos expostos na interface remota e é acionado sempre que um cliente solicita a execução de uma operação no executor do componente [ROM02].

  Este objeto é responsável pela ativação de todos os serviços necessários para a execução do executor do componente antes que o próprio executor do componente seja acionado, para isso, o objeto EJB intercepta as requisições entre o cliente e o executor do componente e toma as ações devidas para, em seguida, acionar a operação pedida [ROM02].
- **Interface remota**: Interface onde são declaradas todas as operações que podem ser executadas em um Executor do Componente. Esta interface é usada pelo

contêiner para criar o “objeto EJB” que interage com o cliente para o acionamento das operações do Executor do Componente.

- **Objeto *Home***: O objeto *Home* é o elemento de software responsável pelo controle do ciclo de vida de um executor de componente e conseqüentemente do componente como um todo. É instanciado a partir de uma classe gerada pelo contêiner com base na interface *Home*. O objeto *Home* é responsável pela criação, destruição ou, no caso de EJBs de entidade, pela localização de um objeto executor de componente. Um *Home* é criado para atender a um único executor de componente.
- **Interface *Home***: Interface onde são declaradas as operações de controle de tempo de vida, que serão implementadas pelo objeto *Home*.
- **Chave primária (*Primary key*)**: Classe que representa a chave primária de um EJB de entidade. É usada nos métodos "*finder*" do objeto *Home* para localizar uma determinada instância do executor do componente no contêiner.

As implementações para as classes que declaram o funcionamento do objeto *Home*, do objeto EJB e do objeto chave primária são criadas pelo contêiner durante o processo de instalação, portanto elas fazem parte do contêiner e seu funcionamento respeita as especificações do modelo de componentes.

A customização do funcionamento do componente e a definição do conjunto de serviços que fará uso no contêiner é feita em arquivos de configuração que seguem o padrão XML. Estas configurações são chamadas “descritores de instalação” (*deployment descriptors*).

2.3.1.1 MODELO DE EXECUÇÃO PARA COMPONENTES EJB

O acionamento de EJBs (com exceção dos componentes EJB orientados e mensagens) é realizado por meio dos métodos de construção (*create*) ou de localização (*finder*) implementados no objeto *Home*. Estes métodos são acessados remotamente pelo cliente para a: criação, localização ou destruição de um componente.

Para o cliente obter a referência ao objeto *Home*, é preciso instanciar um objeto *InitialContext* (*javax.naming.InitialContext*), referenciado no texto daqui para diante, como: “contexto JNDI”.

O contexto JNDI é o ponto de partida para a resolução de nomes, é a partir dele que se obtém qualquer referência a objetos dentro de um contêiner EJB, usando como chave de pesquisa um “apelido” que é passado como parâmetro para o serviço JNDI disponível no contêiner [SUN02].

A execução de um componente EJB pode ser resumida em sete passos:

1. **Obtém a referência para o objeto *Home***: A identificação do objeto *Home* dentro do JNDI é possível devido a um apelido dado ao objeto *Home* no processo de instalação (descrito mais à frente).
2. **Retorna a referência para o objeto *Home*:** Se o apelido informado existe no registro JNDI, a referência ao *Home* é devolvida.

3. **Obtém a referência para o objeto EJB:** Para obter a referência ao objeto EJB, é preciso acionar um método *factory* ou *finder* que resultam em referências a objetos.
4. **Cria o objeto EJB:** O objeto *Home* cria uma nova instância do objeto EJB que implementa a interface remota.
5. **Retorna a referência do objeto EJB:** A chamada para um método *findBy* (*finder*) ou *create* (*factory*), sempre retorna uma referência ao objeto EJB que implementa a interface remota.
6. **Invoca os métodos de negócio:** Aciona os métodos remotos como se fossem locais. A natureza remota do processo é totalmente escondida.
7. **Delega a requisição ao executor do componente:** Depois de contatar o contêiner para acionar os serviços iniciais do computador, a requisição é direcionada para o objeto executor do componente propriamente dito, que fará a execução do método pedido.

A figura 2-6 representa a execução simplificada de um componente EJB (baseado em [ROM02]):

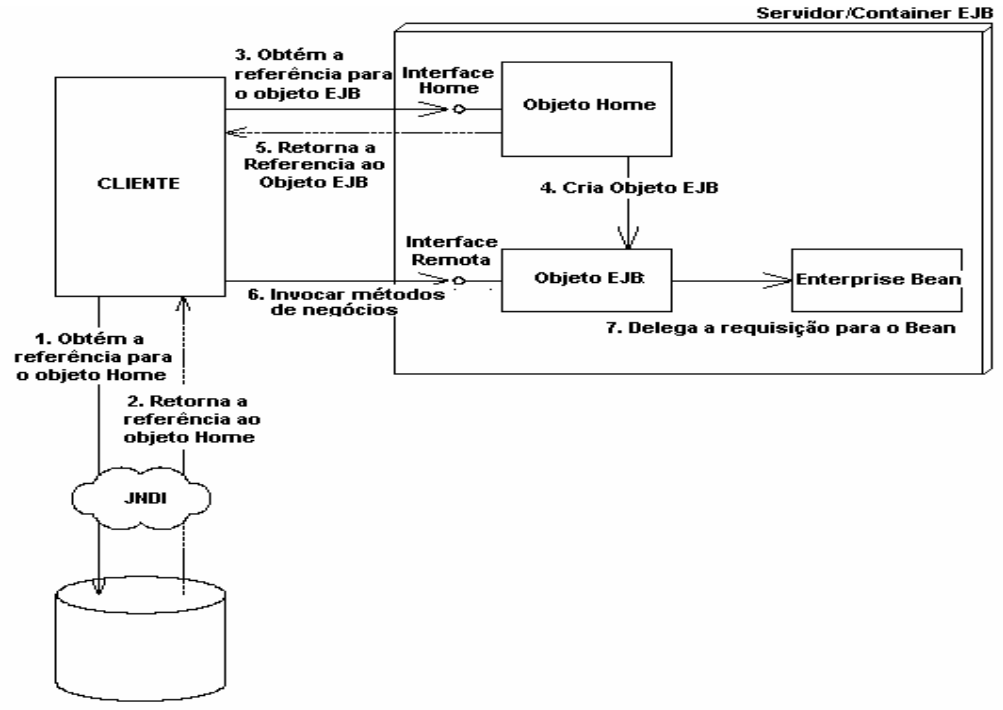

Figura 2-6 - Representação simplificada do processo de acionamento de operações em um EJB.

2.3.1.2 EMPACOTAMENTO E INSTALAÇÃO

Depois de criados os elementos que fazem parte do componente (interface *Home*, interface Remota e o executor do componente), cabe ao desenvolvedor configurar os descritores de instalação.

Os descritores de instalação são configurados em arquivos XML que instruem à ferramenta de instalação o que fazer com os componentes empacotados:

São dois arquivos padrões para componentes:

- **ejb-jar.xml**: Que contém as configurações de cada componente, entre elas:
  - O tipo do componente (*Sessão, Entidade, orientado a mensagens*).
  - O nome da interface *Home* usado pelo componente
  - O nome da interface Remota usada pelo componente
  - O subtipo do componente: sem estado *(stateless),* ou com estado *(stateful*)
  - O nome usado para registro do componente no JNDI
  - O tipo de persistência, no caso de EJBs de entidade (gerenciada pelo contêiner ou gerenciada pelo componente)
  - O nome da classe usada como chave primária, no caso de EJBs de Entidade.
  - Lista de campos usados no caso de EJBs de entidade com persistência gerenciada pelo contêiner
  - Lista de referências entre EJBs (componentes que usam outros métodos de outros componentes)
- **ejb-jar<específico de contêiner>.xml**: Cada implementação de contêiner tem suas particularidades e recursos específicos (eventualmente não comuns a outras implementações) a configuração destes elementos particulares do produto sendo usado são especificados em arquivos de configuração proprietários.

O resultado do empacotamento é a criação de um arquivo compactado pelo utilitário JAR, contendo todos os arquivos que fazem parte dos componentes que se deseja instalar no contêiner, conforme representa a figura 2-7:

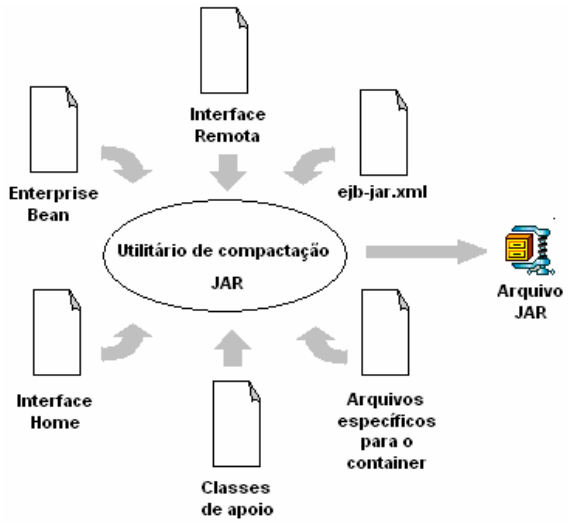

Figura 2-7 - Processo de empacotamento de componentes EJB

A instalação pode ser realizada por diversas ferramentas (usa-se preferencialmente a ferramenta fornecida pelo fabricante do contêiner, que normalmente apresenta facilidades em relação às configurações particulares de sua implementação).

As atividades realizadas por uma ferramenta de instalação são:

1. Descompactação do arquivo JAR
2. Compilação dos arquivos de interface para a geração das classes proxie
3. Registro dos componentes no servidor (registro no serviço de nomes e gravação das classes: do componente – geradas, Executor do componente e classes de apoio - nos diretórios do servidor de aplicação)

A figura 2-8 representa a atividade de um utilitário de instalação:

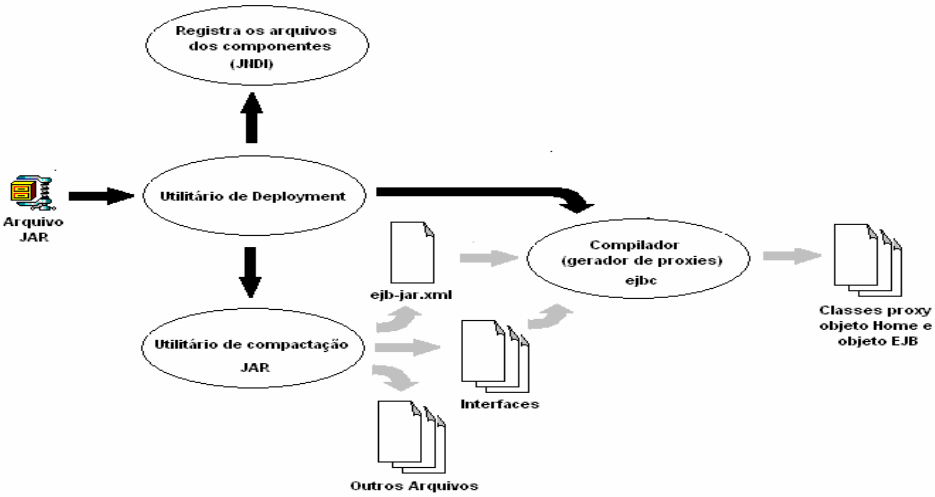

Figura 2-8 - Processo de instalação para componentes EJB

2.3.2 CCM

O modelo de componentes CCM descreve como um componente é constituído (objetos *Home*, executores de *Home*, executores de componentes), como armazena suas propriedades internas (atributos) e como se comunica com outros componentes (portas).

O modelo de componentes CCM apresenta uma forma mais flexível de classificação de "tipos de componentes" que o modelo EJB. A classificação dos componentes CCM deve ser observada sobre alguns aspectos:

1. **Quanto ao "nível" do componente:** A especificação CCM identifica dois níveis de componentes:
   - **Componentes Básicos**: Mecanismo para tratar objetos CORBA tradicionais já existentes como componentes permitindo que objetos CORBA já existentes possam facilmente ser convertidos em componentes.
   - **Componentes Estendidos**: Apresentam características sofisticadas (em relação aos componentes básicos), entre elas, a padronização da forma de acesso às funcionalidades do componente por meio de "portas" e da

alteração na especificação dos atributos (que podem ser especificados para permitir o lançamento de exceções nos métodos modificadores e leitores) para customização das propriedades do componente.

2. **Quanto ao "ciclo de vida" do componente**: Para que o contêiner que hospedará os componentes CCM possa garantir a execução (do componente) da forma esperada pelo desenvolvedor, as instruções relativas ao seu funcionamento são descritas em um arquivo de configuração (usando a linguagem CIDL - *Component Implementation Definition Language* – Linguagem de definição de implementação de componente), a linguagem CIDL prevê 5 categorias de componentes:

   - **Serviço (*Service*)**: É o equivalente CCM aos componentes EJB de sessão sem estado (*Session Bean Stateless)*. Este tipo de componente tem seu tempo de vida limitado a duração da execução do método chamado pelo cliente. São componentes usados para chamadas de métodos pontuais como consultas ou execução de cálculos.
   - **Sessão (*Session*):** É o equivalente CCM aos componentes EJB de sessão com estado (*Session Bean Stateful)*. Seu estado é preservado durante toda a existência do cliente (sessão) ou até que este cliente solicite explicitamente a destruição do componente. Um componente do tipo "sessão" mantém seu estado (valores de atributos e controle de operações em execução) de forma independente para cada cliente. Este tipo de componente é útil para aplicações que usam operações do tipo iterador ou quando o resultado de uma operação chamada é um pré-requisito para o acionamento da operação seguinte.
   - **Processo *(Process)***: Componentes persistentes usados para representar, no servidor, processos de negócio em lugar de representar tuplas em tabelas de bancos de dados.
   - **Entidade (*Entity*)**: É o equivalente CCM aos componentes EJB de entidade. Este tipo de componente é persistente (continua existindo mesmo depois que o cliente que o criou tenha sido destruído). Tem identificação formal dentro do contêiner e podem ser re-obtidos através de uma chave primária (execução de operações *find* nos *Homes* que os gerenciam). Seu principal uso é para realizar mapeamento objeto relacional, desta forma, cada instância representa na forma de objetos uma determinada tupla de uma tabela (ou eventualmente formada por mais tabelas).
   - **Vazio (*Empty*)**: Não tem equivalente nos demais modelos de componentes, e a sua existência é mais um elemento que comprova a versatilidade do modelo. Um componente "vazio" tem seu funcionamento totalmente customizado pelo desenvolvedor através dos descritores de instalação e da implementação dos executores.

O modelo EJB apresenta um tipo específico de componente para a manipulação de mensagens assíncronas, o modelo CCM prevê que qualquer componente CCM pode lançar e "escutar" eventos na forma de mensagens assíncronas usando de forma indireta o serviço de notificações CORBA.

Um componente CCM é uma estrutura de software composta de um conjunto de objetos CORBA, alguns gerados pelo desenvolvedor e outros pelo contêiner a partir da leitura dos arquivos CIDL e dos descritores de instalação.

Na nomenclatura do modelo CCM os elementos que tornam possível a execução das operações nos componentes são chamados "executores". Os executores estão para os componentes assim como os "*servants*" estão para os objetos CORBA [OMG02][OMG02b].

A declaração de um componente é feita na forma de arquivos de definição de interfaces:

- **Arquivos de Definição de Interfaces**: Como todo objeto CORBA um componente CCM também deve ter as suas interfaces declaradas em um arquivo de definição de interfaces. Trata-se de um arquivo que usa a versão estendida da linguagem IDL. Durante o processo de compilação das interfaces para geração das classes de apoio, as definições escritas em IDL estendidas são convertidas para IDL equivalente (IDL 2.x) antes de serem interpretadas para a efetiva geração dos *proxies* (representados na figura 2-9 como "*skeletons"*):

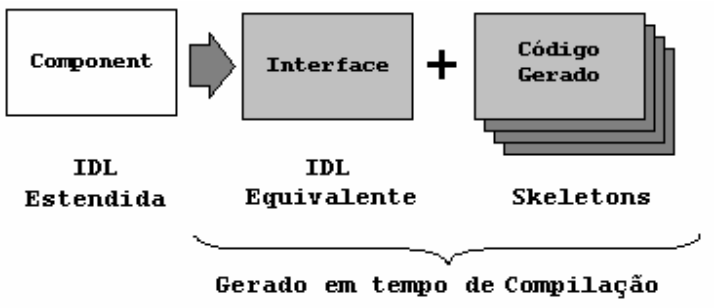

Figura 2-9 - Compilação de IDL Estendida

- **Arquivos de Definição de Implementação do Componente (arquivo CIDL):** Os arquivos CIDL são escritos pelo desenvolvedor e usados para gerar os elementos internos que permitem a integração do componente com o contêiner.

Os programas que implementam as interfaces geradas após a compilação são classificados como:

- **Executor do *Home* (Objeto *Home*)**: Objeto gerado pelo contêiner a partir da compilação do arquivo CIDL. O executor do *Home* é o código que é efetivamente executado quando o *Home* do componente recebe uma chamada. É o equivalente ao objeto *Home* do modelo de componentes EJB.

  Trata-se de um objeto executado no lado servidor, usado pelo cliente para criar ou destruir instâncias de componentes (associar ou desassociar referências aos objetos que implementam as interfaces do componente).

  Do ponto de vista do cliente, o *HOME* é o objeto CORBA usado para gerenciar as instâncias de componentes de um determinado tipo. Cada tipo de componente tem pelo menos um *HOME* para controlar seu ciclo de vida:

  - Criar instâncias de componentes;

- Localizar instancias (e obter sua referência);
- Remover instâncias

Os objetos *Home* escondem do desenvolvedor, os detalhes de gerenciamento de tempo de vida do componente dentro do contêiner. São dois tipos de objetos *Home*:

- **Componentes sem chave primária (*keyless*)**: Não são associados com chaves primárias.
- **Componentes com chave primária (*keyful*)**: São componentes com chave primária, quando a instância do componente é associada a uma chave primária. A chave primária é um valor do tipo *Components::PrimaryKeyBase* que identifica um componente de forma exclusiva (como a chave primária de um tabela identifica um determinado registro).

As operações de instanciação de novos componentes do tipo gerenciado pelo *Home* são operações de "*factory*" [GAM00][OMG02]. As operações que localizam instâncias já existentes de componentes e devolvem as referências para os clientes são chamadas de "*finder*" [GAM00][OMG02].

Todos os "objetos *Home*" têm que ter ao menos uma operação de *factory* padrão. Em componentes que usam *Home*s com chave primária, além da operação de *factory*, é esperada uma operação de *finder* padrão (localização de instâncias de componentes). Estas operações padrões não precisam aparecer na declaração da Interface IDL estendida do *Home*, por que são criadas implicitamente durante a geração das interfaces em IDL equivalente.

- **Executor do componente:** Trata-se da implementação propriamente dita do componente. É análogo ao executor do componente do modelo EJB.
- **Executores do componente para interfaces remotas:** Objetos gerados pelo contêiner a partir da leitura do arquivo CIDL e dos descritores de instalação. Eles implementam as interfaces que são acessadas pelo cliente e quando recebem uma requisição, a direcionam para o executor do componente (ou acionam alguma operação no contêiner). São elementos análogos aos objetos Remotos do modelo EJB.

Uma característica inexistente nos demais modelos abordados neste documento é o conceito de "porta". Componentes CCM contam com uma rica variedade de opções para comunicar-se com o meio externo (clientes com componentes e componentes entre si), estas "opções" de comunicação são chamadas de "portas" e podem ser de 4 tipos:

- **Facetas**: São as diferentes interfaces oferecidas por um componente para aceitar a execução de operações síncrona de clientes ou de outros componentes. A idéia da faceta é permitir que um componente possa oferecer mais de uma interface permitindo que os clientes o acessem sob vários ângulos diferentes (como as diferentes facetas de um diamante). Desta forma, um componente CCM pode oferecer várias formas de uso para cada tipo de componente.[OMG02][MAR00].

Não se deve esquecer que um componente CCM é composto de objetos CORBA, desta forma, cada faceta (que também é um objeto CORBA) é acessada através de sua própria referência ao objeto. O contêiner CCM oferece meios para que o cliente verifique quais referências a objetos pertencem a qual instância e de qual componente.

- **Receptáculos:** Componentes são unidades de composição, e para tanto, o modelo CCM declara portas que representam a associação entre componentes. Receptáculos identificam quais facetas de outros componentes podem ser acessadas. Para o modelo CCM, quando um receptáculo recebe a referência a uma faceta, diz-se que houve uma "conexão".
- **Receptoras de Eventos:** Portas que atuam como pontos de conexão, para onde eventos de um determinado tipo podem ser direcionados. As implementações para este tipo de porta são capazes de obter o que foi emitido por uma fonte de eventos (uma porta emissora de eventos).
- **Emissoras de Eventos**: Portas responsáveis pela emissão dos eventos gerados em um determinado componente para os canais de eventos oferecidos pelo contêiner.

Para representar uma "instância" para o componente, o modelo CCM estabelece que cada componente deve oferecer (além de suas facetas) uma interface chamada de "interface equivalente" (que não necessariamente descreve operações de negócio). Quando um cliente obtém a referência à interface equivalente de um componente, diz-se que obtém a referência para o componente. É através desta referência que um cliente pode acessar as portas.

A figura 2-10 representa o modelo abstrato de um componente CCM, apresentando as diversas portas descritas acima, baseado em [OMG02]:

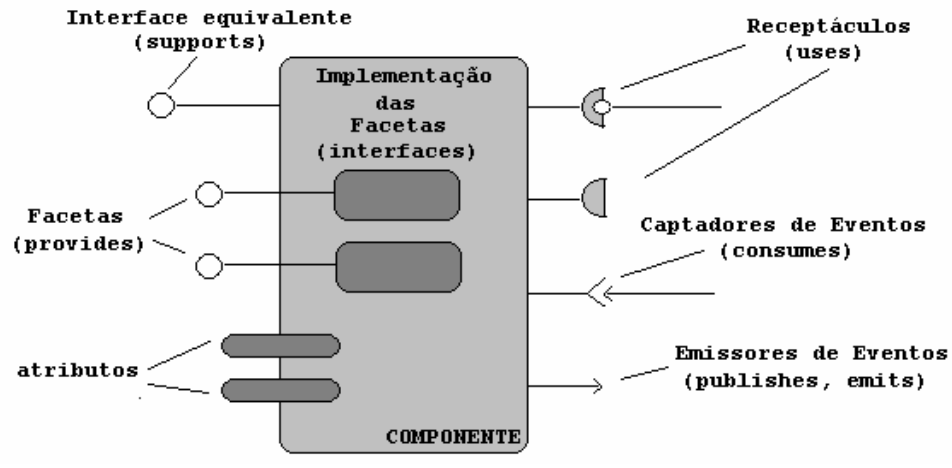

Figura 2-10 - Modelo abstrato para componentes CCM

O conceito de "porta" não se aplica a componentes "básicos", a única característica visível na figura 2-10 suportada por componentes deste tipo são os atributos. Um atributo de um componente CCM é similar a um atributo de um objeto CORBA, mas permite o lançamento de exceções a partir de suas operações modificadoras (*set*) e leitoras (*get*).

2.3.2.1 MODELO DE EXECUÇÃO PARA COMPONENTES CCM

O acionamento de componentes CCM é realizado por meio dos métodos de construção (*factory*) ou de localização (*finder*) implementados no objeto *Home*. Estes métodos são acessados remotamente pelo cliente para a criação, localização ou destruição dos componentes.

Existem dois tipos de clientes que podem fazer acesso a componentes CCM:

- **Clientes que não "enxergam" o componente (*unaware*):** O acesso à referência do componente é realizada da mesma forma que a obtenção de um objeto CORBA tradicional usando, por exemplo, o serviço de nomes (*naming service*). Este tipo de cliente não tem conhecimento da existência do *Home*, pode, portanto, executar somente as operações contidas nas interfaces "suportadas" pelo componente (declaração "supports"). [OMG02]
- **Clientes que "enxergam" o componente e seus recursos (*aware*):** Quando o cliente "sabe" que a interface sendo acionada pertence a um componente CCM. Neste caso o cliente acessa o componente por meio do *Home* e tem acesso a todas as operações disponíveis no componente, como por exemplo: recursos de navegação entre as suas diversas facetas.

O processo de acionamento de um componente a partir de um cliente "aware", em geral segue os passos apresentados a seguir:

1. **Obtém a referência para o objeto *Home***: O acesso às interfaces do componente é conseguido por meio do serviço de nomes do *contêiner*, obtido diretamente do ORB (operação: *resolve_initial_references*), este processo é análogo ao processo de obtenção do Objeto *InitialContext* no modelo EJB.
2. **Obtém a referência para o executor do componente:** O acesso ao executor do componente é realizado de forma indireta (via POA), mas para isso, o cliente deve ter a referência do objeto que aponta para o executor. Essa referência é conseguida como resultado de uma operação *factory* ou *finder*.
3. **Solicita ao POA a criação ou ativação de um executor para o componente:** A criação de uma nova instância do executor ou a utilização de uma instância já existente para atender a requisição depende exclusivamente do adaptador de objetos (POA).
4. **Retorna a referência ao executor do componente:** O resultado de uma operação *create* ou *find* é uma referência ao componente que indiretamente referencia o executor.
5. **Invoca os métodos de negócio:** Aciona os métodos remotos como se fossem locais. A natureza remota do processo é totalmente escondida.
6. **Passa para o executor:** O contêiner recebe a requisição e a trata (por meio das classes geradas em tempo de instalação, que fazem a interface entre o componente e o contêiner).

Os passos descritos estão representados na figura 2-11:

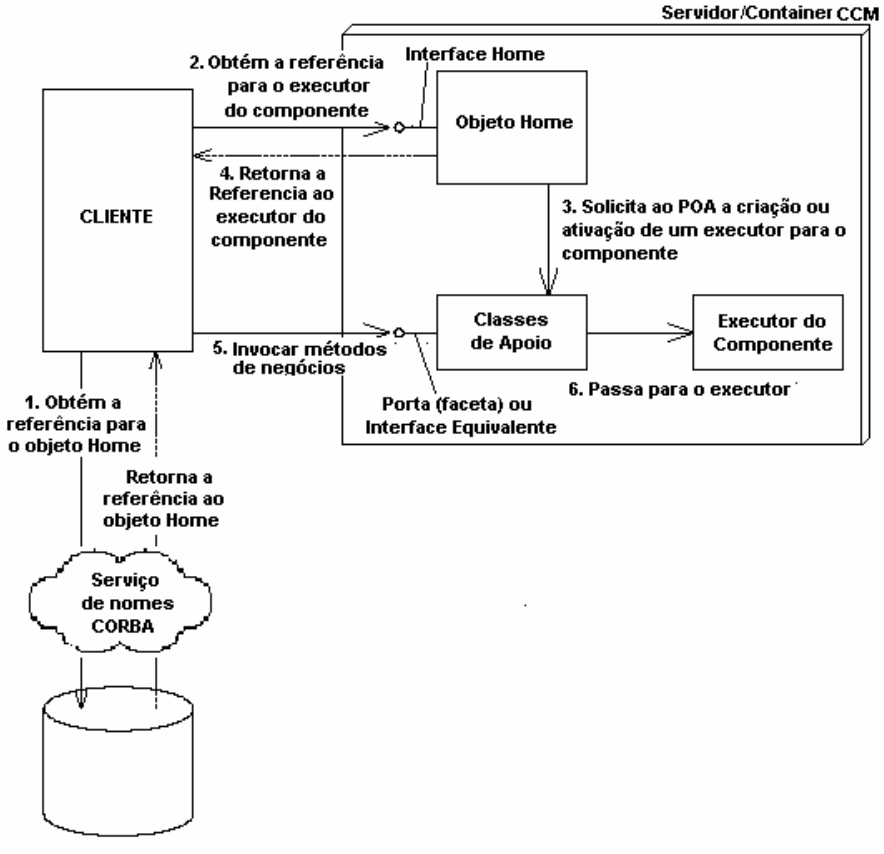

Figura 2-11 - Processo de acionamento de componentes CCM.

A forma como os componentes são executados é customizada pelo implementador por configurações definidas em tempo de instalação e registrados em arquivos de configuração.

Entre os elementos que alteram o funcionamento de um componente, estão: o tipo de API selecionada, o modelo de uso CORBA escolhido e a política de tempo de vida do *servant (servant lifetime policy)* que pode ser uma das seguintes: método (*method*), transação (*transaction*), componente (*component*) e contêiner.

2.3.2.2 EMPACOTAMENTO E INSTALAÇÃO

O processo de empacotamento de componentes CCM é um pouco diferente do processo usado para empacotar componentes EJB. No caso do CCM, são criados pacotes para a instalação de componentes e de conjuntos de componentes. Cada um destes pacotes contém um conjunto de arquivos de descritores (arquivos contendo instruções para que o utilitário de instalação possa realizar o trabalho de publicação da aplicação, de forma que cada componente tenha seu funcionamento como esperado pelo desenvolvedor).

A instalação é realizada por um utilitário específico, que faz a leitura dos arquivos que compõem o(s) componente(s) a partir de arquivos compactados no formato ZIP, compostos por:

- **Descritores de pacote**: São dados mantidos em arquivos escritos usando-se um vocabulário XML próprio contendo as características do pacote e referências aos demais arquivos contidos no pacote.
- **Demais arquivos**: Arquivos que seguem no mesmo pacote. Os tipos destes arquivos dependem do tipo de pacote criado.

A ferramenta de instalação lê o pacote, o descompacta, obtém os descritores e realiza a instalação propriamente dita.

O pacote é o arquivo que contém todos os arquivos necessários para a efetiva instalação e registro do componente no servidor. Existem dois tipos de pacotes usados por uma aplicação CCM, um que acondiciona somente arquivos referentes a um componente e seu funcionamento, e outro que se refere a um conjunto de componentes e seus relacionamentos entre si. A estrutura de construção dos pacotes de instalação para o modelo CCM é resumida na figura 2-12:

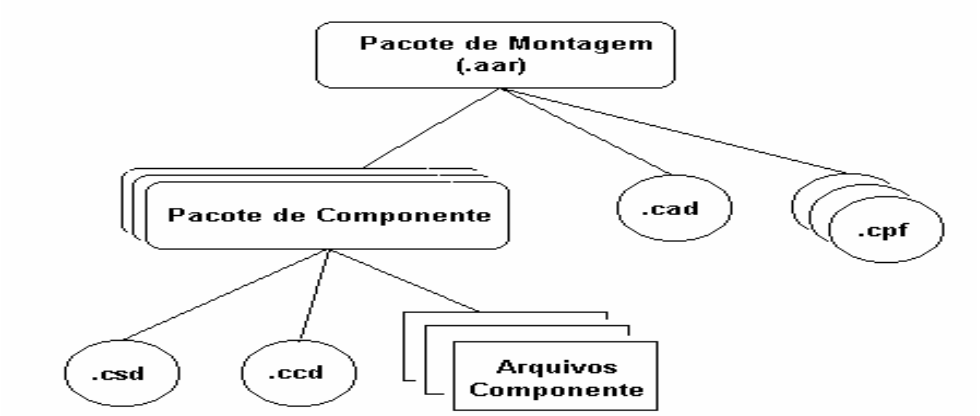

Figura 2-12 - Formato de um pacote CCM

Pacote de Componente (*Component Package*):

Este pacote agrupa os arquivos de um componente. É o arquivo usado para realizar a instalação de um único componente. Pacotes de componentes contam com os seguintes descritores:

- **Descritor de pacote**: Arquivos com extensão ".csd", que contém informações gerais sobre a implementação do software. Um arquivo ".csd", antes de o pacote ser gerado, deve ser acondicionado em uma pasta chamada *meta-inf*. O descritor de pacote faz referência ao descritor do componente.
- **Descritor do componente**: Arquivo com extensão ".ccd", gerado pelo compilador CIDL que contém informações sobre o funcionamento dos componentes dentro do servidor (tipo de componente, propriedades, atributos, tipos de serviços que usa, enfim: informações normalmente descritas nos arquivos CIDL).

Além dos descritores, o pacote de componente inclui os arquivos que compõem o componente: componentes, *Home*s e *helpers*.

Pacote de Montagem (*Assembly Package*):

Arquivo compactado no padrão ZIP, nomeado com a extensão ".aar". É um arquivo que agrupa um conjunto de pacotes de componentes. É usado para realizar a instalação de mais de um componente. É um molde ou padrão para instanciar um conjunto de componentes e apresentá-los uns aos outros.

Pacotes de montagem contam com os seguintes descritores:

- **Descritor de montagem de componentes** (*Component assembly descriptor*) - Arquivo com extensão ".cad" contendo os elementos usados na montagem, as informações sobre conexões entre as interfaces e parcionamento de componentes (conjuntos de instancias de componentes podem ser livres ou parcionados para conjuntos de computadores hospedeiros e processos). Os arquivos ".cad" contém referências aos arquivos ".csd". Cada pacote de montagem tem um único arquivo ".cad", que deve ser colocado em uma pasta chamada *meta-inf* antes do pacote ser gerado.
- **Arquivos de propriedades**: São arquivos criados com a extensão ".cpf" que são usados por programas "configuradores" em tempo de instalação para detalhar atributos de *Home*s ou de instâncias de componentes.

2.3.3 MTS/ COM+

O modelo de componentes COM+ (ou modelo de componentes MTS) descreve como um componente COM+ (ou componente MTS) deve ser construído.

Quanto à classificação, componentes MTS são componentes transientes (que não mantém o seu estado após a finalização da vida do cliente que o acionou). Quanto ao controle transacional, componentes MTS podem ser [ROF99] [BOX98]:

- **Transacionais**: Componentes que tem as suas classes COM de implementação (classe onde as operações do cliente são efetivamente executadas) configuradas como: "*Requires a transaction*" ou "*Requires a new transaction*". Classes COM configuradas desta maneira são *stateless* (mantém o seu estado somente durante uma única chamada de operação). Isso acontece porque o servidor MTS cria uma nova instância do componente a cada final de transação, garantindo que o estado do objeto COM não se propague além dos limites da transação (respeitando as características ACID).
- **Não Transacionais**: Componentes que têm suas classes COM configuradas como "*Does not support transactions*", componentes com classes COM definidas desta forma podem ser programados para funcionar como os componentes do tipo *sessão* do modelo CCM e de sessão com estado (*session stateful*) do modelo EJB. Mantém o seu estado enquanto o cliente que o acionou existir.
- **Baseados em interface**: Os objetos receptores de eventos implementam uma interface contendo as operações que serão usadas pelos emissores de eventos. Quando um evento é disparado, o emissor faz chamar uma das operações da interface implementada pelo receptor de eventos.

- **Baseado em métodos**: Funciona da mesma forma que o "baseado em interface" com a diferença que os métodos são definidos diretamente, sem a necessidade de uma interface exclusiva para este fim.
- **Persistente**: Implementa o modelo de funcionamento "publica/assina" (tem funcionamento igual aos componentes baseados em mensagens do modelo EJB que operam na modalidade : "publica/assina" – publish/subscribe). Neste caso, nem a fonte de eventos nem o receptor de eventos conhecem o "parceiro". A fonte de eventos publica os eventos que podem acontecer e o receptor de eventos "assina" os eventos que deve "receber".

O modelo de componentes MTS incorpora o serviço MSMQ (Microsoft *Message Queue*) que permite que o "enfileiramento" de mensagens assíncronas (especialmente no caso de eventos do tipo "persistente") até que o "receptor de eventos" possa atendê-las. A Implementação da troca de mensagens acontece internamente no componente, não existe um tipo de componente específico para a troca de mensagens assíncronas como ocorre no modelo EJB.

2.3.3.1 O COMPONENTE MTS

No modelo MTS o componente é formado por diversos elementos que interagem para garantir a sua efetiva execução, entre estes elementos destacam-se [KIR97][KIR97a]:

- **Factory do Objeto**: Implementação das operações da interface IClassFactory acessada pelo MTS executive por meio do "factory wrapper", no momento da criação de uma nova instância do componente.
- **Objeto MTS**: É um objeto COM+ instalado no servidor. O objeto MTS é o equivalente ao objeto EJB do modelo EJB e ao executor do modelo CCM, trata-se do elemento de software que efetivamente executa as operações solicitadas pelo cliente.

Um componente MTS é, antes de tudo um objeto COM, por isso os elementos que compõem um objeto COM também compõem um objeto MTS, assim, o processo de criação de um componente MTS inicia na codificação do arquivo de interface (IDL), passando pela sua compilação, a implementação de operações de negócio, empacotamento (na forma de DLL – um requisito específico para componentes MTS) e finalmente, a instalação.

O modelo de componentes simplifica o processo de codificação para o desenvolvedor. Quando se desenvolve um objeto COM tradicional, é preciso definir um conjunto de artefatos relacionados ao seu acionamento e gerenciamento de ciclo de vida, direcionando parte do esforço de programação para atividades não relacionadas com a implementação de rotinas que atendam o negócio, tais como:

- ***Factory***: O modelo de objetos COM, exige a codificação de uma classe *factory* que sabe como criar objetos de um determinado tipo. No caso de componentes MTS, a implementação deste *factory* é responsabilidade do próprio MTS.
- **Registro de DLL:** Componentes COM devem oferecer informações de registro. As interfaces são descritas por meio de IDL (MS-IDL, não é a mesma

linguagem do padrão CORBA) que gera *proxy/stub* para a DLL e bibliotecas de tipos.

- **Contagem de referências**: Objetos COM tradicionais devem tornar disponível a interface *IUnknown* para atualizar o contador de referências, usado para gerenciamento de ciclo de vida de objetos.
- **Interface de consulta**: O cliente acessa também a interface *IUnknown* para obter a *QueryInterface*, através da qual acessa os recursos de um objeto COM.
- **Interface de distribuição (*IDispatch*):** Para ser acessado por programas escritos em linguagens de script (como ASP ou outras normalmente usadas para geração dinâmica conteúdo para Web), ele deve implementar a interface *IDispatch*.
- **Interfaces para pontos de conexão (*IConectionPoint*)**: Para que um componente possa emitir eventos e para que um cliente possa lança-los, componentes e clientes devem implementar a interface *IconnectionPoint*.
- **Informações sobre tipos**: Gerados a partir dos arquivos de definição IDL.
- **Métodos:** Implementação propriamente dita das operações do objeto.

Quando se desenvolve componentes usando-se o modelo COM+, estes elementos são fornecidos pelo ambiente de execução (contêiner), devido a características semelhantes de funcionamento que a grande maioria dos componentes COM+ tem, a figura 2-13 representa esta característica, baseado em [KIR97].

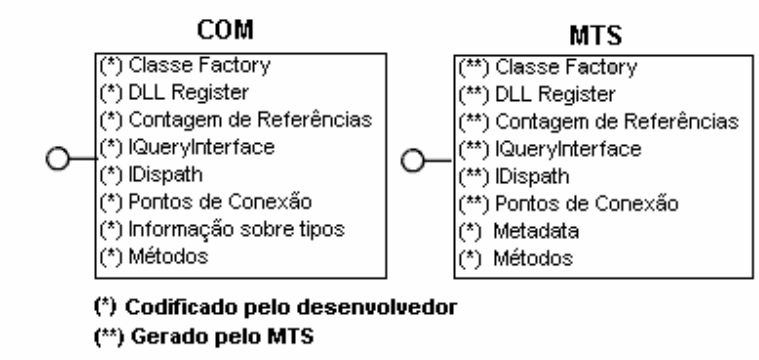

Figura 2-13 Comparação entre os modelos COM e MTS

Os desenvolvedores implementam somente as operações para manipular lógica e prover informações descrevendo as características da classe, sem preocupar-se com os detalhes relacionados com aspectos como: controle de ciclo de vida, segurança ou tratamento de mensagens.

Outra modificação em relação ao modelo COM é que as interfaces podem ser definidas diretamente na linguagem de programação escolhida pelo desenvolvedor, sem o uso de IDL. A descrição padrão de interfaces é feita através de metadados gerados por utilitários. Cada funcionalidade oferecida por um componente pode ser descrita em uma interface diferente (como acontece com as facetas dos componentes CCM).

2.3.3.2 MODELO DE EXECUÇÃO PARA COMPONENTES MTS

O processo geral de acionamento de componentes MTS pode ser resumido nos 7 passos numerados em seguida:

1. **Cria uma nova instância do componente**: Na verdade, o objeto (implementação das operações de negócio do componente) não é instanciado até que o contêiner receba a execução de um método do componente. Quando o MTS *executive* recebe uma requisição, ele seleciona qual dos *wrappers* irá manipula-la. O componente não é instanciado até que uma chamada do cliente alcance o contêiner. Quando o cliente solicita a criação de um componente, ele chama o construtor e devolve a referência ao objeto.
2. **Obtém a referência do componente**: A criação do componente devolve para o cliente uma interface do tipo IUnknown convertida (cast) para o tipo esperado pelo cliente.
3. **Solicita a execução de uma operação**: Assim que a requisição é recebida pelo servidor, o contêiner a intercepta e designa um adaptador (*wrapper)* [GAM00] para atende-la, este executa o algoritmo de gerenciamento de acionamento de instâncias (JITA). Para selecionar se vai efetivamente instanciar um novo objeto MTS ou usar um disponível no pool.
4. **Direciona a chamada para o respectivo objeto MTS**: Instancia o objeto MTS no servidor
5. **Atualiza a situação do contexto**: Quando a requisição é atendida, a resposta enviada de volta ao cliente, e o componente então executa as operações *SetComplete* / *SetAbort* ou *Release*, o componente é destruído.
6. **Devolve o resultado da operação**: O resultado da operação é encaminhado para o *wrapper*.
7. **Resultado da operação**: O resultado da operação é encaminhado para o cliente.

A figura 2-14 ilustra o processo de acionamento descrito:

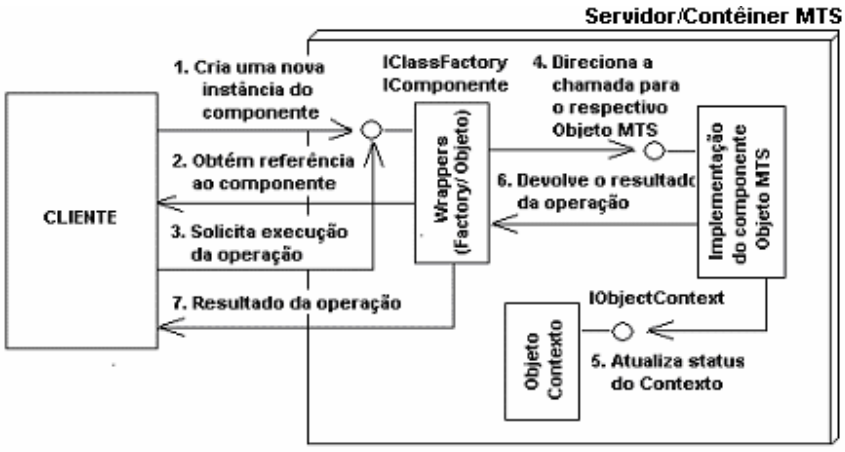

Figura 2-14 - Processo de acionamento de operações em um servidor MTS

O acionamento de componentes MTS é realizado da mesma forma que o acionamento de objetos locais, de forma transparente. Assim que a requisição do cliente é recebida, esta é direcionada a um dos *wrappers* do componente, *factory* ou *object*.

2.3.3.3 EMPACOTAMENTO E INSTALAÇÃO

Componentes MTS usam DLLs como forma de empacotamento para os componentes.

1. Escreve-se o código fonte das classes;
2. O Compilador da linguagem usa os serviços de compilação do MTS para criar os metadados que descrevem a classe (no lugar de criar manualmente um arquivo IDL);
3. O Compilador da linguagem também gera os binários da classe;
4. Chamadas a outros serviços serão inseridos na classe pelo compilador, baseado nas palavras chaves e atributos especificados no fonte;
5. O código binário da classe e os *metadados* são usados pelo "*linker*" para gerar pacotes de classes na forma de DLL;

Um resumo do processo é representado na figura 2-15:

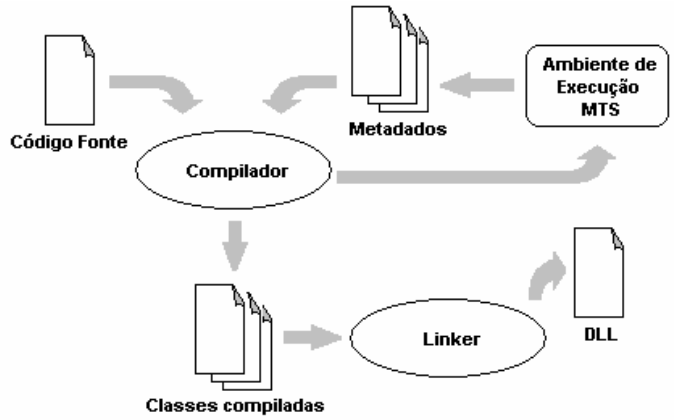

Figura 2-15 - Criação de uma DLL para instalação de componentes MTS

A instalação é realizada utilizando-se softwares com interface gráfica bastante intuitiva, fornecida pela Microsoft.

2.4 COMPARAÇÃO ENTRE MODELOS DE COMPONENTES

Componentes são independentes de suas interfaces o que garante que a modificação no seu código seja totalmente transparente para os elementos de software que usam aquele componente. Além de oferecerem uma "lista" de operações disponíveis para execução em um componente, interfaces são usadas para a geração de elementos de software de apoio (classes *Helper*) com as quais os clientes efetivamente interagem para ter acesso às operações do componente e com as quais o componente interage para obter os serviços do contêiner.

Modelos de componentes definem APIs que devem ser usadas durante o desenvolvimento de software, o que garante que um componente desenvolvido sob um determinado modelo possa ser instalado em qualquer contêiner que respeite o modelo, de forma transparente.

Estes aspectos permitem que o desenvolvedor foque sua total atenção para questões envolvidas com o negócio (o que a aplicação deve fazer), sem a preocupação de fazer com que os módulos do sistema dêem suporte aos recursos oferecidos naturalmente pelos modelos de componentes.

A tabela 2.3 relaciona algumas características oferecidas pelos modelos de componentes abordados neste trabalho:

Tabela 2-3 - Comparação entre os modelos de componentes.

CCM	MTS	EJB
TIPOS DE COMPONENTES		
Dispõe de 4 tipos: serviço, sessão, processo e entidade. Um quinto tipo de componente é descrito na especificação como "EMPTY". Tipo que permite ao desenvolvedor construir um tipo de componente que atenda a necessidades *ad-hoc*.	Existe somente um tipo de componente que tem seu funcionamento customizado para atender as necessidades da aplicação: Comparado ao modelo CCM, quando componentes MTS são configurados para serem transacionais, eles funcionam como um componente CCM do tipo "serviço".	O modelo EJB dispõe de 3 tipos de componentes: EJB de sessão (que podem ser "sem estado" ou "com estado", *Entity Beans* e *Message Based Beans*:
FLEXIBILIDADE DE FUNCIONAMENTO DOS COMPONENTES		
O modelo CCM conta com contêineres baseados em POA (Adaptador portável de objetos). Adaptadores Portáveis de Objetos são configuráveis por um conjunto de políticas definidos no momento de sua criação. Os tipos de componentes previstos no modelo criam POAs (contêineres) com conjuntos pré definidos de políticas que são ajustadas a partir de dados contidos em arquivos CIDL, garantindo uma vasta possibilidade de funcionamento para componentes CCM.	Componentes MTS têm o seu funcionamento alterado de acordo com configurações realizadas na ferramenta usada para a instalação que alteram a forma como atua o processo *MTS executive* que atende àquele componente quanto a requisitos não funcionais como: controle de transações, utilização do pool, etc...	O modelo EJB conta com um grupo fechado de tipos de componentes que operam sempre da mesma forma (dependendo do tipo especificado).
FORMA DE ACESSO		
Para obter a referência a um componente, inicialmente obtém-se a referência ao objeto *Home* do Componente e solicita-se sua criação ou localização (se for um componente persistente).	Componentes MTS são acessados de forma totalmente transparente (entre os modelos, é o mais intuitivo para o programador). O ambiente é responsável por interpretar quais são requisições	Componentes EJB são acessados inicialmente por meio de seu objeto *Home* que retorna a referência ao objeto EJB que realiza a interface entre o cliente e a implementação propriamente dita

	remotas e quais são locais. O servidor MTS cria adaptadores (*wrappers*) que efetivamente interagem com a implementação do componente.	do componente. A referência ao objeto *Home* é obtida por meio do serviço de nomes (JNDI) do servidor. De posse da referência ao objeto *Home*, operações "*create*" ou "*finder*" obtém a referência a componentes criados ou recuperados do serviço de persistência.
SUPORTE A TROCA DE MENSAGENS ASSÍNCRONAS		
A troca de mensagens é realizada no modelo CCM por meio de PORTAS receptoras e emissoras de eventos. Estas portas interagem com o serviço de mensagens do contêiner.	Componentes MTS dão suporte a mensagens por meio do MSMQ (*Microsoft Message Queue* – Fila de Mensagens da Microsoft). O componente MTS pode atuar como emissor (publicador – *publisher*) ou receptor (assinante - *subscriber*).	O modelo EJB especifica um tipo de componente para dar suporte a mensagens. Este componente interage com o JMS (*Java Message Service*) assumindo o papel de consumidor de mensagens.

2.5 CONCLUSÕES

Quando se desenvolve uma aplicação sobre um modelo de componentes, o projeto é direcionado para a definição das interfaces de cada componente e da sua conexão com os demais. A configuração dos serviços usados por um determinado componente deixa de ser uma preocupação para o desenvolvedor.

Durante o projeto, declara-se: as interfaces dos componentes, as conexões que podem ser estabelecidas entre eles e o comportamento esperado para cada componente.

O comportamento é declarado na forma de "tipo" de componente (um tipo de componente tem funcionamento pré-estabelecido) como ocorre com EJB ou ajustando-se parâmetros de execução como no modelo MTS ou CCM.

Componentes são elementos de software potencialmente reusáveis que podem ser instalados em ambientes diferentes contanto que estes ambientes tenham o contêiner para que possam ser executados, o que permite que um componente desenvolvido para uma determinada aplicação possa ser acessado por outras aplicações.

A limitação principal está no fato de que a aplicação que usa um componente deve ser compatível com o modelo de componentes, ou seja, quando se projeta um software orientado a componentes se leva em consideração o ambiente de execução do componente.

3 ESPECIFICAÇÂO DO MODELO DE COMPONENTES CORBA

O objetivo deste capítulo é oferecer um resumo de alto nível da especificação CCM para que o leitor possa entender as decisões de projeto aplicadas ao contêiner CCM descrito no capítulo seguinte.

A especificação do modelo de componentes CORBA descreve um *framework* para a confecção de aplicações baseadas em componentes usando-se CORBA, em especial, padroniza a forma como deve ser feito um servidor CORBA e quais interfaces devem ser oferecidas pelo servidor e pelos objetos que fazem parte de um componente, escondendo do desenvolvedor os pormenores ligados à arquitetura CORBA como: o registro de objetos no serviço de nomes ou o controle de instâncias de objetos CORBA disponíveis em memória.. O *framework* é organizado em:

- **Modelo de componentes**: Descrição dos elementos que formam um componente abordando o ponto de vista do desenvolvedor de componentes e das interfaces expostas aos clientes.

  O modelo de componentes ou "modelo abstrato de componentes" como era chamado na primeira versão da especificação [OMG99a], descreve as portas (facetas, receptáculos, emissores de eventos e receptores de eventos) e como elas são declaradas para componentes básicos e estendidos.

  O funcionamento dos *Homes* também é descrito nesta parte da especificação assim como a fase de configuração de componentes.

- **Semântica e sintaxe da CIDL:** A linguagem CIDL (*Component Interface Definition Language – Linguagem de definição de Interface do Componente)* é usada para declarar os pormenores sobre o funcionamento interno do componente, como o nome do capítulo descreve, trata-se da descrição da linguagem propriamente dita, um catálogo de palavras chave.

- ***Framework* de implementação CCM:** Descreve como a linguagem CIDL deve ser usada para descrever a composição do componente: categoria do componente, qual programa (ou programas) assume a função de executor do componente e de executor do *home*.

  É também neste capítulo que o mapeamento entre IDL estendida e IDL equivalente é descrito assim como as regras para geração das interfaces locais.

- **Modelo de programação do contêiner:** Descreve as interfaces que compõem as APIs usadas pelos componentes para interagir com os clientes (APIs externas), as definições das características de cada categoria de componente, as APIs usadas para cada categoria de componente, a forma como os serviços são oferecidos aos componentes e as variações de comportamento esperadas para cada categoria de componente.

- **Integração com EJB**: O modelo de componentes CORBA pode ser entendido como um super-conjunto do modelo de componentes EJB, o capítulo descreve a forma padronizada de como componentes CCM acessam componente EJB e vice-versa.

- **Empacotamento e instalação:** Descrição de como os descritores de instalação devem ser confeccionados e como os componentes devem ser empacotados para poderem ser instalados em outros contêineres. Este assunto já foi explorado na seção 2.3.2.2.
- **XML DTDs:** Descrição das regras para o uso de elementos que são incorporados aos descritores de instalação. Como os descritores de instalação do modelo de componentes CORBA são escritos em XML, a descrição dos "domínios" de cada elemento que compõe o documento é feita na forma de DTD.
- **Meta-modelo do Repositório de Interfaces:** A especificação representa as interfaces criadas para o modelo de componentes CORBA na forma usando MOF. Como a IDL do modelo de componentes é uma extensão da IDL básica (CORBA), o capítulo descreve inicialmente a IDL CORBA e em seguida a IDL usada no modelo de componentes.
- **Meta-modelo do *Framework* de implementação CCM:** Contém a representação MOF do framework de implementação CCM.

Os assuntos: "Semântica e sintaxe CIDL", "XML DTD" são catálogos contendo a sintaxe usada respectivamente nos arquivos de configuração e nos descritores de instalação.

Os metamodelos "do repositório de interfaces" e "da implementação CCM" oferecem a representação de como os elementos que compõem o framework CCM, na verdade trata-se da especificação na forma de digramas.

Não faz sentido explorar "catálogos" de interfaces ou listas de marcadores XML neste documento, desta forma, este capítulo oferece um resumo do "Modelo de componentes", do "Framework de implementação CCM", do "Modelo de programação do contêiner" e da "integração com EJB" que efetivamente representam a parte central da especificação.

3.1 MODELO DE COMPONENTES

O modelo declara dois níveis de componentes: básico e estendido. Declara também as portas que podem ser oferecidas por um componente e informa que a identidade de um componente é conseguida através da obtenção da referência à sua interface equivalente que é obtida por meio de um objeto *Home* que pode usar ou não chaves primárias para este fim. Estes aspectos já foram explorados na seção 2.3.2.

As potencialmente diversas interfaces expostas pelo componente, chamadas de facetas, são declaradas em IDL estendida. Cada uma delas é implementada por um objeto CORBA e cada um destes objetos CORBA é considerado parte integrante do componente e como tal, seu ciclo de vida respeita o ciclo de vida do componente.

De posse da referência a uma faceta se pode obter a referência à interface equivalente de um componente e o contrário também é verdadeiro. Também se deve identificar quando duas referências a objetos pertencem à mesma instância do componente.

Isso significa que a coesão entre as instâncias dos vários objetos que compõem uma instância de um componente deve ser controlada pelo contêiner.

3.1.1 PORTAS E O MECANISMO DE NAVEGAÇÃO ENTRE PORTAS

A interface equivalente de um componente CORBA especializa a interface *Components::Navigation* declarada na API. Esta interface oferece operações para a obtenção de referências a todas as facetas declaradas para o componente e para verificar se uma determinada referência a um objeto é na verdade uma porta que pertence à mesma instância do componente testado.

Componentes conectam-se entre si por meio de suas portas, um componente CCM se conecta à faceta de outro componente, estabelecendo uma conexão entre o seu receptáculo e a faceta do componente conectado.

O receptáculo representa a conexão estabelecida entre componentes (um receptáculo armazena a referência à porta do componente conectado).

Receptáculos podem ser simples (*simplex*) ou múltiplos (*multiple*). Receptáculos simples indicam que existe um relacionamento entre dois componentes, ou seja, quando uma conexão é estabelecida, ela é feita com exclusivamente para uma faceta de cada vez. Já os receptáculos múltiplos podem representar a conexão do receptáculo com diversas facetas.

Quando a declaração da porta é convertida para IDL equivalente, diversas operações são criadas especificamente para estabelecer ou desfazer conexões entre um receptáculo e uma ou mais facetas, além disso, as interfaces equivalentes dos componentes CORBA especializam a interface *Components::Receptacle* que contém operações genéricas para estabelecer ou desfazer conexões e permitir algo semelhante ao que existe com a navegação entre facetas.

Além das portas que oferecem conexões síncronas, componentes CCM oferecem portas assíncronas: emissoras e receptoras de eventos através das quais, objetos que representam “eventos” são trocados entre os componentes por meio de um canal de eventos oferecido pelo contêiner.

As portas emissoras de eventos podem ser: publicadoras (*publishers*) ou emissoras (*emitters*). A diferença entre elas é que as portas “publicadoras” permitem vários “assinantes” e as “emissoras” permitem a conexão de somente um “assinante” por vez.

Quanto ao funcionamento interno, a especificação estabelece que um canal de eventos deve ser oferecido pelo contêiner exclusivamente para encaminhar os eventos criados pela porta “publicadora”, para portas “emissoras”, um canal de eventos pode ser compartilhado entre várias portas.

De forma semelhante ao que acontece entre facetas e receptáculos, para que a porta receptora de eventos de um componente possa receber os eventos publicados por um outro componente deve ser estabelecida uma conexão entre elas. Também de forma semelhante às portas síncronas, o componente oferece operações específicas (geradas em IDL equivalente) e operações genéricas (herdadas da interface *Components::Events*)

Os anexos 1 e 2 oferecem respectivamente as listagens da declaração de um componente em IDL estendida e em IDL equivalente, no anexo 4 foi reproduzido o código fonte do *servant* que implementa as interface equivalentes do componente contendo as

operações declaradas em IDL equivalente e as operações herdadas de *Components::Events*, *Components::Receptacles* e *Components*::*Navigation*.

3.1.2 HOMES

As referências aos componentes CCM são obtidas por meio de operações *factory* [GAM00] expostas pelas interfaces *Home*.

Uma interface *Home* pode ter operações definidas implicitamente ou explicitamente. Cada conjunto de operações está contido em interfaces que são especializadas pela interface *Home*.

As interfaces implícitas contêm operações declaradas automaticamente pelo modelo de componentes no momento da conversão de IDL estendida para equivalente. Para *Homes* que não trabalham com chave primária, sempre é declarada uma operação *create* que não recebe parâmetros e devolve uma referência à interface equivalente do componente.

Para os componentes com chave primária, os *Homes* são declarados com operações específicas para o controle de persistência: *create*, *find_by_primary_key, remove* e *get_primary_key*. As três primeiras operações recebem como parâmetro a chave primária, entre elas, as duas primeiras operações retornam a referência à interface equivalente do componente, *remove* não tem retorno.

A operação *get_primary_key* é usada pelo cliente para obter a referência à chave primária da instância do componente cuja referência é passada como parâmetro.

As interfaces explícitas consistem de operações declaradas pelo desenvolvedor do componente, que explicitamente deseja ter operações de localização ou criação diferentes das operações padrão oferecidas implicitamente pelo contêiner.

Além destas operações os *Homes* implementam as operações herdadas das interfaces declaradas na API dependendo do tipo de *Home*.

3.1.3 CONFIGURAÇÃO DO COMPONENTE

Componentes podem ser configurados (por meio de descritores de instalação) para contemplar (ou não) duas fases no ciclo de vida do componente: configuração e operacional.

Espera-se que a fase de configuração seja usada para definir o estado inicial de um componente identificando os valores iniciais de seus atributos, estes valores devem ser atribuídos durante a inicialização de cada instância daquele tipo de componente. Segundo a especificação, a configuração de um componente deve ser realizada sem a necessidade de instanciar o componente.

Quando um componente é ajustado para efetivamente ter as duas fases, este passa a atender requisições somente depois que a operação "*configration_complete"* é executada através da interface equivalente do componete. O acionamento desta operação identifica o final da fase de configuração e o início da fase operacional.

O framework de implementação CCM oferece mecanismos para permitir desabilitar a execução de operações durante a fase de configuração ou de operação.

Os valores usados na configuração do componente podem ser declarados no arquivo de propriedades (.cpf) que são usados durante a instalação.

3.2 MODELO DE PROGRAMAÇÃO DO CONTÊINER

O modelo de programação do contêiner define como devem ser codificados os componentes, as interfaces, que devem ser implementadas, como os componentes devem ser empacotados e a arquitetura do contêiner onde os componentes CCM serão instalados.

Segundo o modelo de programação do contêiner, o ambiente de execução de um componente CCM oferece:

- **Interfaces Externas:** Interfaces que são visíveis para os clientes. As classes que atendem a estas interfaces são criadas em tempo de instalação (*deployment*) e a sua implementação é feita pelo próprio contêiner, dado que o funcionamento interno destas classes é específico para as necessidades de cada produto. Neste texto, as interfaces externas são também chamadas de interfaces remotas.
- **Interfaces com o Contêiner:** Estas interfaces existem para que o componente possa interagir com o contêiner e vice-versa, são implementadas pelo desenvolvedor. São dois tipos de interfaces com o contêiner:
  - **Interfaces internas:** Interfaces usadas para que o componente possa obter informações sobre os serviços de segurança e de transações e solicitar ações a estes serviços. O acesso aos serviços mencionados é realizado através do contexto (tipo de objeto explicado posteriormente no texto).
  - **Interfaces de *callback***: Interfaces cuja implementação é oferecida pelo desenvolvedor e que são usadas pelo contêiner para acionar operações no componente.

  Existem duas versões para as interfaces internas e de *callback*, estas duas versões são expostas em APIs:
  - **API de sessão (*session)***: Interfaces implementadas em componentes transientes.
  - **API de entidade (*entity)*:** Interfaces implementadas em componentes persistentes.
- **Interfaces Home:** Interfaces que declaram os objetos *Home* são dois grupos:
  - **Interfaces *Home* remotas:** Aquelas que são obtidas pelos clientes para que possam solicitar ao *Home* que forneça uma referência ao componente. As interfaces *home* são internas e o código de seus *servants* é gerado durante a instalação do componente no contêiner (depende da implementação de cada contêiner).
  - **Interfaces *Home* locais:** Geradas em tempo de compilação IDL, mas implementadas pelo próprio desenvolvedor. A especificação não descreve o motivo para a criação destes executores locais, dado que a função do *Home* é obter a referência a um componente e que isso é feito através da interface *Home* remota (que é implementada pelo próprio contêiner garantindo assim

o controle interno consistente das referências aos objetos CORBA que formam um componente).

Um possível uso para objetos que implementam estas interfaces seria o acesso local entre componentes instalados no mesmo contêiner, mesmo assim, a especificação não oferece informações sobre a conexão existente entre os executores das interfaces remotas e os executores locais de *Homes*.

Para acessar os serviços necessários para seu funcionamento, os componentes interagem com o contêiner. No caso de componentes CORBA, virtualmente qualquer serviço CORBA pode estar disponível, especialmente: Nomes, Notificação, Transações, Segurança e Persistência.

O acesso aos serviços é feito indiretamente através dos objetos gerados pelo contêiner durante a instalação. A geração destes objetos acontece à partir da avaliação das definições de funcionamento especificadas em arquivos de "Definição de Implementação de Componentes" (CIDL), e nos descritores de instalação.

A figura 3-1 (baseada em [OMG02]) representa uma visão simplificada de como os principais elementos participam para que um componente possa ser executado em um contêiner CCM:

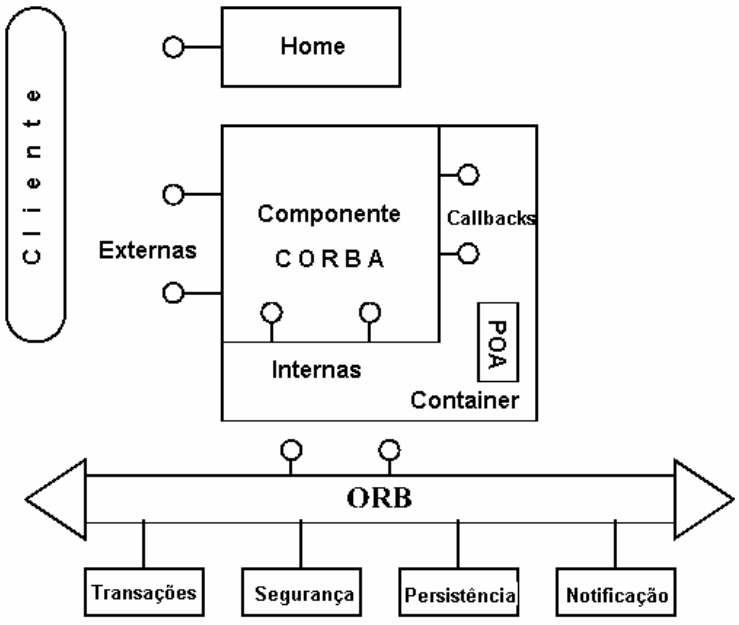

Figura 3-1 - Modelo de arquitetura para um contêiner CCM

Na nomenclatura do modelo CCM os elementos que efetivamente executam as operações para os componentes são chamados "executores" [OMG02]. Os executores estão para os componentes assim como os "*servants*" estão para os objetos CORBA usados com POA [OMG02b][PIL99]. *Servants* são objetos de linguagem de programação que o POA usa para direcionar as requisições baseadas no identificador de objetos contidos na chave do objeto [OMG02b] [PIL02] [ORF98].

O Modelo de uso CORBA especifica como o mapeamento entre identificação de objetos e os *servants* deve ser tratado pelo POA usado pelo contêiner, mais especificamente pelo *servant locator* [OMG02b] [PIL02] [ORF98]. A especificação descreve três modelos de uso CORBA:

- **Sem estado (*Stateless*):** A referência ao objeto é transiente e o *servant* pode ser mapeado a mais que uma identificação de objeto.
- **Com estado (*Conversational*):** A referência ao objeto é transiente e o *servant* é mapeado a somente uma identificação de objeto.
- **Durável (*Durable*):** A referência ao objeto é persistente e o *servant* é mapeado a somente uma identificação de objeto.

Além do modelo de uso CORBA e da API usada, componentes persistentes podem usar chave primária para localizar instâncias ou não. Dadas estas características, o modelo de componentes CORBA oferece basicamente cinco (5) categorias de componentes: serviço, sessão, processo, entidade e vazio. A tabela 3-1 (baseada em [OMG02]) relaciona as categorias de componentes CORBA:

Tabela 3-1 - Categorias de componentes CCM

CATEGORIA	USA CHAVE PRIMÁRIA	MODELO DE USO CORBA	TIPO DE API
Serviço	Não	Sem estado	sessão
Sessão	Não	Com estado	sessão
Processo	Não	Durável	entidade
Entidade	Sim	Durável	entidade

Se nenhum dos tipos de componentes relacionados na tabela 3.1 atenderem às necessidades do desenvolvedor, a especificação prevê uma quinta categoria de componentes chamada “vazio” (*empty*), para componentes totalmente customizados à partir dos descritores de instalação.

O modelo de uso CORBA identifica parte do comportamento esperado de um componente, desta forma se poderia supor que todos os componentes de uma mesma categoria se comportam da mesma forma. Mas, o comportamento de um componente CORBA é influenciado pela política de gerenciamento do tempo de vida do *servant*, um conjunto de políticas que identificam por quanto tempo o POA deve manter uma identificação de objetos ligada à instância de um *servant* que atende a um determinado cliente.

São quatro políticas possíveis e a opção por uma destas “políticas” influencia no comportamento (funcionamento interno) do *servant locator* e conseqüentemente do componente:

- **Método (*Method*):** O componente é ativado sempre que uma operação é acionada e é destruído ao final da sua execução.

  Entenda-se como “ativação do componente” o processo executado no primeiro *pre_invoke* do *servant locator* que é responsável por obter uma

instância do *servant* (uma nova ou uma já disponível no *pool*), liga-la ao mapa de objetos ativos (mantido internamente pelo próprio *servant locator*), instanciar um contexto, atribuir o contexto ao *servant* (*set_session_context*) e acionar a operação de *callback ccm_*activate.

Ao final da execução da operação, o *servant* não precisa necessariamente ser removido da memória, ele pode ser disponibilizado para um pool de *servants*. A sua real destruição (remoção da memória) acontece atendendo aos critérios adotados na implementação do contêiner.

- **Transação (*Transaction*):** O componente é ativado quando a primeira operação que pertence a uma transação é acionada e é destruído quando a transação é encerrada.
- **Componente (*Component*):** O componente é ativado na execução da primeira operação (do componente), mantendo-se ativo até que o próprio componente solicite destruição ou que seja colocado em estado passivo (estado em que permanece com a identificação do objeto que representa a interface equivalente desassociada de qualquer referência a objetos, ou seja, embora instanciado, fica incapaz de receber e atender requisições).

  Depois de recebida a solicitação, o contêiner decide quando efetivamente de alterar o estado do componente para passivo. Quando aplicada esta política, o critério de quando alterar o estado do componente para passivo passa a não ser mais responsabilidade exclusiva do contêiner, que espera a requisição de alteração do estado do componente para passivo (essa requisição pode ser feita pelo próprio componente através do contexto) ou a destruição do componente.
- **Contêiner:** O componente é ativado na execução da primeira operação, mantendo-se ativo até que o contêiner determine a modificação do seu estado para passivo. O critério pra a alteração de estado é implementado pelo desenvolvedor do contêiner.

  A alteração para o estado passivo pode acontecer mesmo em componentes que não precisam persistir seu estado. Para aumentar a escalabilidade do contêiner, adota-se algum critério para remover da memória *servants* inativos há muito tempo e que ainda não foram destruídos ou que ainda não receberam solicitação, através do contexto, para se alterar seu estado para passivo. Isso acontece quando a política de tempo de vida do *servant* adotada é mais extensa que uma transação (como nos casos: *component* e *contêiner*).

Componentes da categoria "serviço" são os que têm seu comportamento mais previsível, a única política de gerenciamento do tempo de vida do *servant* que podem usar é "método", as demais categorias permitem o uso de qualquer uma das políticas listadas.

3.3 *FRAMEWORK* DE IMPLEMENTAÇÃO CCM

O *framework* de implementação do modelo de componentes CORBA ou CIF (*CCM Implemenmtation Framework*) descreve como a interpretação dos dados contidos nos arquivos CIDL influenciam a geração do código de apoio pelo contêiner.

O principal artefato usado pelo CIF é o arquivo contendo as declarações CIDL (*Component Implementation Definition Language*).

Para o CIF, os elementos que contém a efetiva implementação de cada objeto CORBA que forma um componente são chamados de *executores*. A figura dos executores está para o modelo de componentes CORBA assim como a figura do *servant* está para o POA [BOL02].

Na verdade, falar-se de executores e *servants* é falar sobre os mesmos elementos (dado que, cada objeto que compõe um componente é na verdade um objeto CORBA acionado por um POA).

As regras que detalham a implementação dos componentes e seus *homes* são descritas em arquivos CIDL. Estes arquivos são usados pelo contêiner durante a instalação para gerar o código que implementa as interfaces previstas nas respectivas APIs (a API usada depende do tipo de componente que é especificado no CIDL). Estas implementações geradas oferecem todas as características (e operações) que correspondem ao funcionamento básico de um componente CCM.

Em outras palavras, o conjunto de artefatos gerados automaticamente entre a declaração das interfaces em IDL estendida e a instalação é o "*framework* de implementação de componentes" ou CIF (*Component Implementation Framework).*

A definição de implementação de um componente é feita em CIDL dentro de uma "composição". Cada composição declara:

- O tipo de *home* que gerencia o componente (obtido das declarações em IDL equivalente)
- O nome do executor do *home*: Nome do programa que atuará como executor do *home* (nome do *servant* do objeto *home* que gerencia a vida do componente descrito na composição).
- O nome do executor do componente: Nome do programa que atuará como executor para a interface equivalente do componente (nome do *servant* do objeto que implementa a interface equivalente do componente descrito na composição). Eventualmente são declarados mais que um executor para o componente, os chamados "executores segmentados".

A necessidade de se identificar os executores do *home* e do componente é que eventualmente o desenvolvedor do componente pode optar por usar executores desenvolvidos por si e não usar os gerados pelo contêiner. Estes executores oferecidos pelo desenvolvedor devem ser especializações dos executores gerados pelo contêiner.

Em um arquivo CIDL, o desenvolvedor declara a relação existente entre os vários elementos de um componente assim como o tipo de tratamento que o container deve dar a ele. É esta estrutura que se chama "composição". A declaração mínima de uma composição é mostrada na listagem 3-1:

```
composition <categoria> NomeDaComposicao {
      home executor NomeDoExecutorDeHome{
      implements  TipoDoHome;
      manages NomeDoExecutorDoComponente;
      };
};
```

Listagem 3-1 - Estrutura genérica da declaração de uma composição usando-se CIDL

A figura 3-2 ilustra graficamente a forma mínima que a declaração de uma composição deve assumir (baseado em [OMG02]):

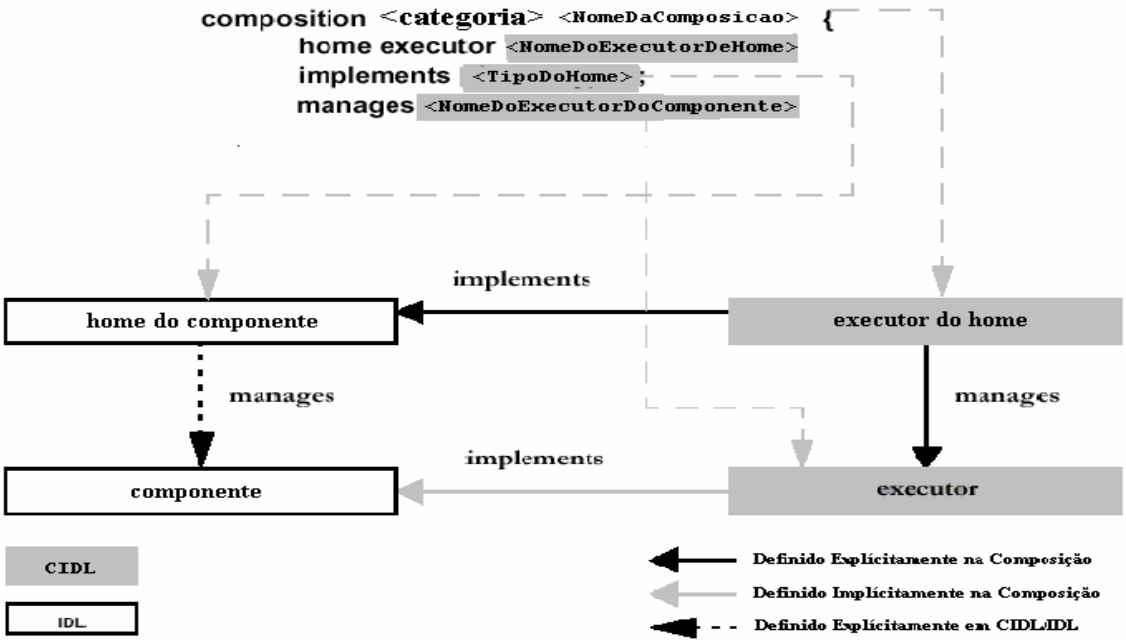

Figura 3-2 - Estrutura mínima de uma composição em CIDL.

As composições declaradas para componentes que implementam a API entidade (categorias: processo e entidade) devem identificar a forma como o gerenciamento de persistência deve ocorrer. O gerenciamento de persistência é oferecido pelo contêiner usando-se indiretamente o PSS (*Persistence State Service*) ou serviço de persistência CORBA [OMG02d].

A declaração de como a persistência deve acontecer a objetos CORBA comuns é feita em arquivos contendo declarações na linguagem PSDL (*Persistent State Definition language*).

Porém, assim como acontece com o serviço de notificações CORBA, o contêiner atua como mediador no acesso ao serviço de persistência por meio das declarações feitas em CIDL, que é um super-conjunto da linguagem PSDL usada pelo modelo de componentes CORBA.

Sendo assim, os conceitos aplicados ao serviço de persistência CORBA são também aplicados ao controle de persistência oferecido pelo modelo CCM, são elas [OMG02d]:

- **Objeto abstrato de armazenamento (*abstract storage type*)**: Declara os atributos que serão persistidos e um nome que, no caso da linguagem Java, é o nome da interface que será criada pelo compilador de PSDL. Além disso, para cada atributo declarado, a interface gerada à partir da declaração incorpora operações de atribuição e obtenção de valores (*setters e getters*).

  PSDL é um superconjunto da linguagem IDL, portanto, a declaração do tipo de armazenamento abstrato pode conter qualquer uma das construções previstas em IDL.

- **Objeto de armazenamento (*storage type*):** Nome da entidade de software que atuará como *servant* do objeto abstrato de armazenamento declarado anteriormente.

- **Home abstrato do armazenamento (*abstract storage home*)**: Declara a assinatura das operações usadas pelo PSS para controlar o ciclo de vida de um objeto abstrato.

- ***Home* do armazenamento (*storage home*):** Nome da entidade de software que atuará como *servant* do *Home* abstrato do armazenamento declarado anteriormente.

- **Catálogo (*catalog*)**: Objeto CORBA local que especializa *CosPersistentState::CatalogBase* através do qual se obtém as referências aos objetos: *home* abstrato do armazenamento e ao *home* do armazenamento. É por meio do catálogo que as atualizações são efetivamente realizadas no dispositivo de armazenamento e nos objetos de armazenamento ligados ao catálogo.

Em CIDL, pode-se declarar um catálogo à ligação da composição com o catálogo e com o *home* do objeto de armazenamento (*storage home*). Quando declarado em CIDL, um catálogo recebe uma "etiqueta" (*label*) que é usado para identifica-lo no restante do documento, como representado na figura 3-3 (baseada em OMG02), na página seguinte.

Sabendo-se que CIDL é um super-conjunto de PSDL, todos os elementos declarados em PSDL podem ser declarados em CIDL, por isso, as declarações dos objetos: *home* do armazenamento e objeto de armazenamento podem ser feitas diretamente no documento CIDL (embora não representado na figura 3-3).

A especificação declara as regras de conversão para as declarações: de chaves primárias, *finders*, *factories* e remove, cada uma destas operações é delegada para a sua equivalente gerada para PSS [OMG02]. Esta regra vale tanto para as geradas implicitamente quanto para as declaradas explicitamente.

No caso das operações explícitas, cada operação do *home* deve ser explicitamente "delegada" para uma operação do *home* abstrato do armazenamento.

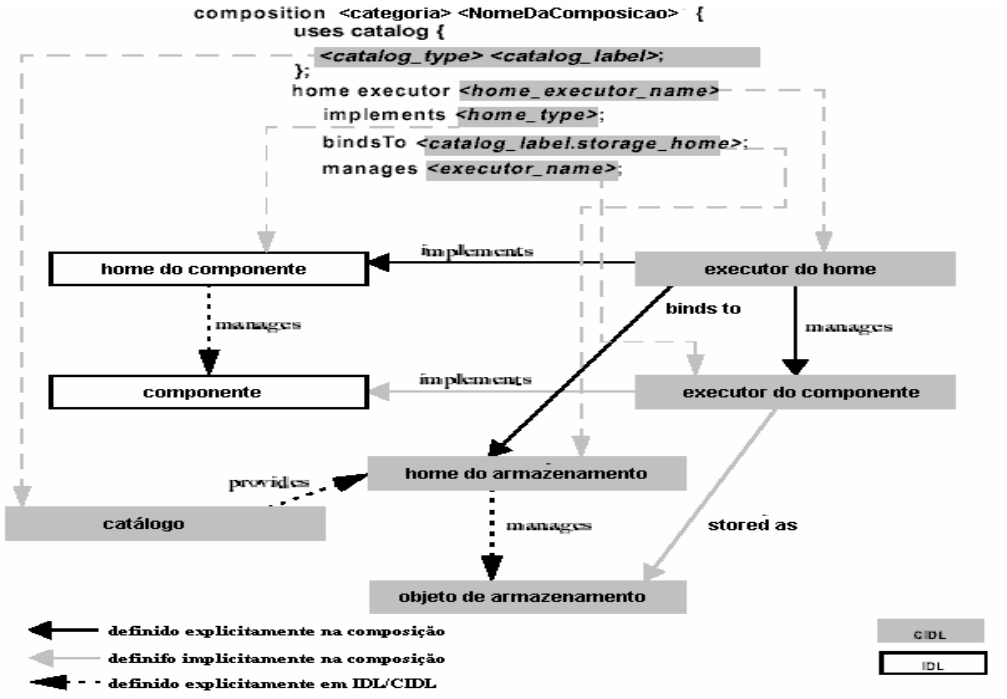

Figura 3-3 – Estrutura de uma composição em CIDL que declara componentes persistentes.

Quanto aos executores do componente, estes podem ser monolíticos ou segmentados, para executores segmentados, deve-se declarar um *home* abstrato de armazenamento para cada segmento que corresponde a uma ou mais facetas que são instanciadas independentemente do restante do componente, permitindo um gerenciamento mais otimizado dos recursos computacionais consumidos (memória e tempo de processador).

Outra característica do CIF é o fato de poder declarar "*proxy homes*" nas composições. Como o nome sugere, são objetos *home* que atuam localmente, mas que delegam a execução de suas operações a objetos localizados remotamente.

3.4 INTEGRAÇÃO COM O MODELO EJB

O modelo de componentes CORBA padroniza a forma como componentes CCM ou clientes CCM devem acessar as operações de componentes EJB e vice-versa.

A técnica usada para permitir a interação entre os dois modelos é o modelo de interação (*interworking model*) descrito na especificação CORBA [OMG02b]. O modelo consiste em declarar *proxies* [GAM00] que na especificação recebem o nome de "visões" am ambos os lados. Quando um objeto CORBA acessa um objeto em outro modelo, o objeto CORBA aciona operações na visão (*proxy*) que representa o objeto remoto.

O acionamento da operação no objeto remoto propriamente dito é feito usando-se um artefato de software que a especificação [OMG02b] chama de "ponte", que, no caso de interação entre objetos, pode ser classificado como um mediador (*mediator*) [GAM00] dado que deve executar a chamada às operações dos objetos em outra tecnologia para objetos distribuídos isolando do objeto CORBA qualquer conhecimento sobre esta ação.

A ponte que liga a visão que o objeto de uma tecnologia tem do outro (em outra tecnologia) deve sofrer uma conversão, fazendo com que a ponte, além de precisar de um mediador tenha que usar um adaptador [GAM00] para converter as chamadas de uma interface para outra.

O conceito aplicado na interação entre componentes CCM e componentes EJB segue o mesmo modelo. A especificação CCM descreve todo o mapeamento que deve ser feito pelo adaptador para que o mediador possa fazer com que a ponte acione operações entre componentes das duas tecnologias.

A figura 3-4 (baseada em OMG02) representa a interação entre clientes (que podem ser componentes), visões e pontes.

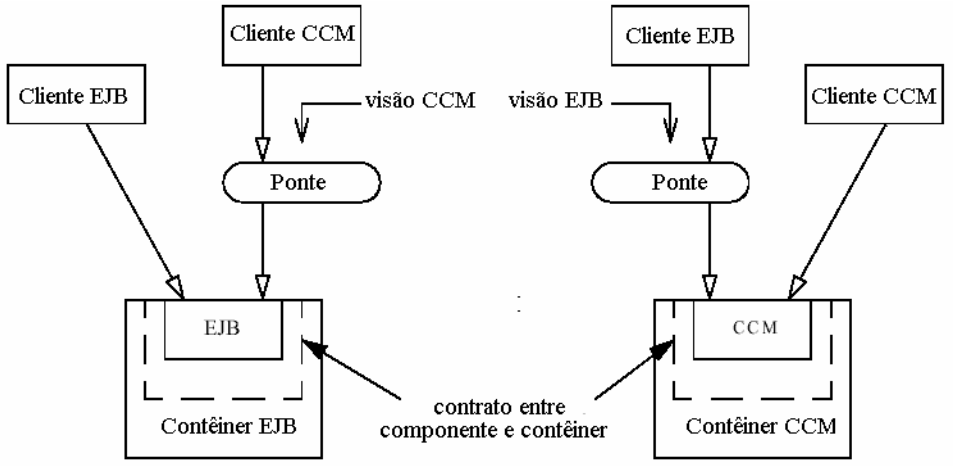

Figura 3-4 - Visão geral dos elementos usados na interconexão entre componentes EJB e CCM.

3.5 CONCLUSÃO

O modelo de componentes CORBA identifica um modelo de componentes totalmente flexível, diferente dos demais modelos que oferecem comportamentos pré-estabelecidos aos componentes, um componente CCM pode ser ajustado para ter qualquer comportamento.

Embora o modelo CCM ofereça características interessantes como: a aplicação do conceito de portas, o estabelecimento de conexões e a navegação entre facetas e receptáculos, a flexibilidade no comportamento do componente é o grande diferencial do modelo em relação aos demais. O comportamento dos componentes CORBA pode ser completamente alterado ajustando-se os atributos dos descritores de instalação para aplicar diferentes modelos de uso, API ou categorias e até mesmo ajustar quais são os executores e sua distribuição (entre ambientes de execução potencialmente diferentes) através das declarações feitas em CIDL.

Contudo, o oferecimento de tal flexibilidade aumenta a complexidade dos arquivos de configuração e do código fonte gerado (se comparado com os modelos EJB e MTS) para executar as interfaces do componente, reduzindo a abstração que o modelo de componentes CORBA deveria oferecer em relação a arquitetura CORBA (para obter maior customização do componente, o desenvolvedor deve estar familizarizado com a arquitetura CORBA).

Quanto a integração entre tecnologias, embora a especificação do modelo de componentes CORBA identifique e padronize a integração com componentes EJB, a interconexão com o modelo MTS (utilizado internamente na arquitetura .NET) é possível usando-se o mecanismo de interconexão "*interworking*" descrito na especificação CORBA 3.0 [OMG02b]. Mais uma característica que diferencia o modelo CCM dos demais.

4 DESCRIÇÃO DE UM CONTÊINER PARA COMPONENTES CCM DO TIPO SESSÃO

Antes de continuar, é importante lembrar que o modelo de componentes CORBA especifica na verdade um *framework* para o desenvolvimento de aplicações CORBA, mas aplicando os conceitos de componentes.

Em outras palavras, um componente CORBA em execução em um contêiner CCM é um conjunto de objetos CORBA que, como tais, devem ter: interfaces declaradas usando-se IDL, ligação com um POA que gerencie seu ciclo de vida, um *servant* que efetivamente executa as operações declaradas e registro em algum serviço que permita aos clientes obterem suas referências.

As interfaces declaradas para estes objetos CORBA especializam as interfaces da API padrão do contêiner de acordo com o papel que assumem no componente (internas, externas ou de *callback)*.

O elemento central do contêiner é o POA, a entidade responsável pela instanciação de cada objeto envolvido e pela execução da operação correta em cada um destes objetos. É responsável também pela criação das referências aos objetos, pela garantia de que os *servants* estejam disponíveis para receber as requisições quando estas chegarem e pelo direcionamento das requisições recebidas aos respectivos *servants* para que sejam atendidas.

O POA intercepta cada requisição destinada a um componente e decide entre: obter um *servant* do *pool*, (se houver um *pool*) remover da memória um *servant* que não recebe requisições há algum tempo, determinar a referência ao objeto, ligar a identificação do objeto a uma determinada instância do *servant*, etc....

No caso do POA que gerencia os objetos CORBA instalados em um contêiner CCM, uma opção de projeto é ajustar as políticas POA para que a aplicação (o contêiner) use um *servant locator* (um programa que é acionado pelo POA antes e depois da execução de cada operação invocada pelo cliente em um *servant*).

Desta forma, a criação dos POAs que atendem a um componente em um contêiner CCM deve contemplar também a criação dos *servant managers* do tipo *servant locator* [OMG02][PIL99]:

A opção pelo uso de um *servant locator* em lugar de um *servant activator* [OMG02][PIL99] se justifica pelo fato de que, usando-se um *servant locator*, cada requisição recebida pelo ORB é interceptada pelo POA e direcionada para o s*ervant locator* antes e depois de ser executada no *servant* (operações de *callback*: *preinvoke* e *postinvoke*) o que permite ao implementador do contêiner estabelecer as suas próprias estratégias de gerenciamento interno como: manutenção e remoção de objetos em memória para reuso (padrão de projetos *evictor* [BOL02]), a atualização de um mapa de objetos ativos próprio e no caso de componentes CCM ou a identificação de qual componente cada objeto CORBA integra.

O *servant locator* em um contêiner CCM é também o elemento que realiza as chamadas às operações de *callback* do componente, obtém a instância de *servant* que atende a um determinado componente e conecta o componente aos serviços CORBA.

A adoção do *servant locator* como elemento central de gerenciamento e controle do contêiner não é a única opção de projeto possível, mas devido às suas características, é a opção mais natural.

O POA que serve aos objetos que compõem componentes do tipo *sessão* deve aplicar as políticas listadas na tabela 4-1 [OMG99a][BOL02][ORF98]:

Tabela 4-1 – Configuração das políticas aplicadas a um POA que gerencia de componentes em um contêiner do tipo "Sessão".

POLÌTICA	VALOR / JUSTIFICATIVA
POLÍTICA DE FUNCIONAMENTO DE THREADS (*THREAD POLICY*)	**ORB_CTRL_MODEL** – O POA é o responsável pela delegação de requisições para *threads*, assim o controle sobre a criação e execução de *threads* fica sob a responsabilidade do *servant locator*. Essa característica é interessante para o contêiner porque a especificação CCM pede que seja possível configurar o gerenciamento de *threads* por meio da *tag "threading"* nos descritores de instalação. Quando o atributo *policy* da tag *threading* tiver o valor "*multithread*", o *servant locator* deve acionar uma nova *thread* para cada requisição, caso contrário, se o atributo *policy* for ajustado para "*serialize*", o *servant locator* deve executar a operação em uma nova instância do executor.
POLÍTICA DE CICLO DE VIDA (*LIFESPAN POLICY*)	**TRANSIENT** – Componentes *session* não precisam manter estado além do período de execução do cliente que criou o componente. (independente da política de gerenciamento de tempo de vida do *servant* adotada).
POLÍTICA DE UNICIDADE DE IDENTIFICAÇÃO DE OBJETOS (*OBJECT ID UNIQUENESS POLICY*)	**N/A** – Não se aplica quando Política de Unicidade de Identificação de Objetos é ajustada para **NON_RETAIN.**
POLÍTICA DE ATRIBUIÇÃO DE IDENTIFICAÇÃO (*ASSIGMENT ID POLICY*)	**USER_ID** – Esta política instrui ao POA que a atribuição das identificações de objetos deve ser feita pela própria aplicação em lugar de ser responsabilidade do POA. A atribuição da identificação do objeto deve acontecer no acionamento de operações *Home* do tipo *factoty*.
POLÍTICA DE ATIVAÇÃO IMPLÍCITA	**NO_IMPLICIT_ACTIVATION** – Não se aplica quando Política de Unicidade de Identificação de Objetos é ajustada para **NON_RETAIN.**

(*IMPLICIT ACTIVATION POLICY*)	
POLÍTICA DE PROCESSAMENTO DE REQUISIÇÕES (*REQUEST PROCESSING POLICY*)	**USE_DEFAULT_SERVANT** – Instrui ao POA que será usado um *servant manager*.
POLÍTICA DE RETENÇÃO DE *SERVANTS* (*SERVANT RETENTION POLICY*)	**RETAIN** – Atributo declarado desta forma para permitir o uso de um *servant locator.*

4.1 ARQUITETURA DO CONTÊINER

O protótipo consiste de um contêiner que implementa um sub-conjunto da especificação CCM que permite a execução de componentes CCM estendidos da categoria sessão, além disso, oferece o serviço de notificações CORBA (para estabelecer as conexões entre as portas emissoras e receptoras de eventos) e o serviço de nomes(os serviços de transação e de segurança não foram incluídos). Para o gerenciamento de tempo de vida do *servant*, o contêiner oferece somente o conjunto de políticas identificado na especificação como “contêiner”.

Os componentes executados no protótipo podem ter qualquer tipo de porta (mesmo as portas usadas para comunicação assíncrona).

O contêiner, no protótipo é um conjunto de POAs e de *servant locators* responsáveis pela execução de cada tipo de objeto que compõe um componente (executor do componente e portas). Estes POAs são criados aplicando-se o conjunto de políticas relacionado anteriormente na tabela 4-1.

Os objetos criados localmente para a execução de operações de *callback (*executores locais de componentes, homes e portas) também compõem o contêiner, assim como os objetos locais que representam o contexto, através dio qual os componentes podem ter acesso aos serviços oferecidos pelo contêiner.

A figura 4-1 ilustra a arquitetura do protótipo:

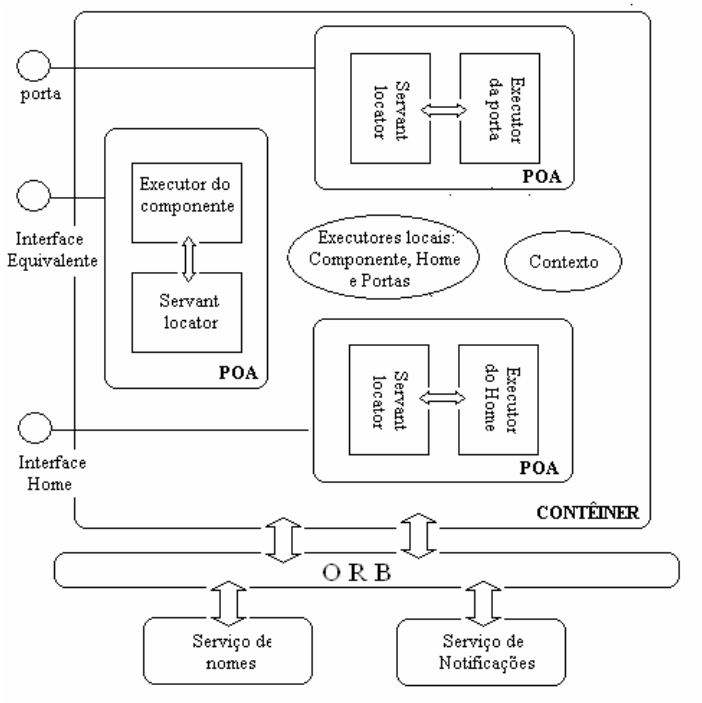

Figura 4-1 - Arquitetura do protótipo

O protótipo não tem utilitários de instalação ou de compilação de IDL e não realiza a leitura de descritores. As informações sobre quais objetos devem ser registrados e gerenciados pelo contêiner foi codificada nas classes: *DeployedHomesSingleton* (que mantém uma estrutura contendo as instâncias de todos os objetos "Home" – um para cada componente) e *SessionConteinerBootStrap* (Demais objetos que formam os componentes).

A classe *CCMServer* assume o papel de "servidor", ela é responsável pela obtenção das referências: ao *ORB*, ao *POA e* ao serviço de nomes. O *CCMServer* é responsável também pelo registro inicial dos objetos que compõem os componentes no serviço de nomes e da respectiva atribuição de identificação de objetos e referências.

Durante a execução do *CCMServer* a classe *DeployedHomesSingleton* é inicializada. Esta classe é um *singleton* [GAM00] que mantém uma lista dos *Homes* dos componentes instalados no contêiner, os dados destes *Homes* (que normalmente seriam obtidos dos descritores de instalação) são oferecidos pela operação "*getHome*" (*private* na própria classe).

Usando os dados obtidos de *getHome*, o *singleton* registra os *homes* nele relacionados para que sejam alcançados por um cliente por meio do "*HomeFinder*"[OMG02].

A especificação [OMG02] informa que todos os *homes* de componentes estendidos devem ser registrados através das operações da interface *HomeRegistration* para que sejam obtidos pelo cliente através da interface *HomeFinder*.

Contudo, a especificação também descreve a interface *HomeRegistration* como: "...uma interface interna que deve ser usada pelo componente para registrar seu *home* ...". Desta forma, infere-se que um componente deve registrar seu próprio *home*.

Dado que existem informações sobre os *homes* nos descritores de instalação e nos arquivos de definição de interfaces, a sua inicialização e registro podem ser realizados pelo próprio contêiner, isolando do desenvolvedor do componente mais um pormenor sobre o funcionamento interno do modelo de componentes.

A solução adotada no protótipo foi realizar o registro automático dos *Homes* sem usar a interface *HomeRegistration*. A opção não fere o funcionamento de componentes que originalmente tenham sido criados para serem executados em uma outra implementação pelo fato de ser uma interface interna do contêiner.

Como o protótipo não oferece inicialização à partir dos descritores de instalação a obtenção de instâncias dos *homes* foi feita na inicialização do "*DeployedHomesSingleton*" sem o uso de pontos de inicialização (entry points). Um ponto de inicialização é um objeto que contém uma operação estática que retorna a instância do executor de um determinado tipo de *home*.

Sobre pontos de inicialização, vale citar um pormenor: no exemplo disponível na especificação, o ponto de inicialização do *home* é colocado no executor do componente (é uma operação do executor), mas no final do mesmo documento, a especificação determina que o ponto de inicialização pode ser uma operação disponível em qualquer classe que atue como *factory* [GAM00] para o *home.* Além disso, deixa claro que a identificação do ponto de inicialização é uma declaração opcional no descritor de instalação permitindo o seu uso ou não.

O *CCMServer* também inicializa o contêiner. Uma atividade realizada pela classe *SessionConteinerBootStrap* que cria os POAs e *servant locators* e registra os objetos que formam cada componente instalado no serviço de nomes.

Para cada um destes objetos, um POA com o conjunto de políticas descrito na tabela 4-1 é criado e um *servant locator* ligado a ele. As referências a objetos são criadas e as identificações de objetos são atribuídas, mas os *servants* não são instanciados aplicando-se o padrão de projetos "ativação preguiçosa" (*lazy activation*) [BOL00].

A figura 4-2 representa a seqüência de operações realizadas ao se executar o *CCMServer*:

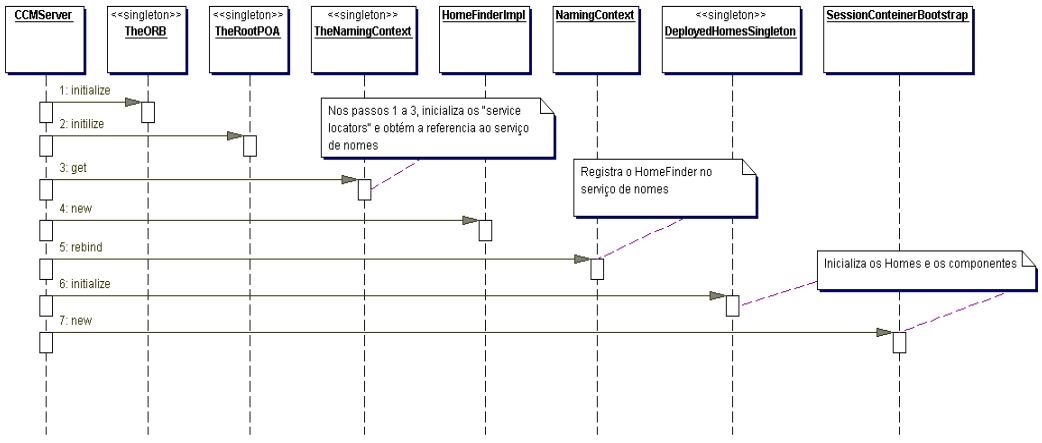

Figura 4-2 - Diagrama que representa a execução do *CCMServer*

Uma vez completada a seqüência representada na figura 4-2, os componentes estão prontos para receber requisições de um cliente.

Para que um cliente possa executar operações em um componente CCM, este deve obter uma referência ao *Home* que gerencia o componente. Para tanto, usa-se o *HomeFinder* serviço cuja referência é obtida através da operação "*resolve_initial_references*" do ORB passando-se como parâmetro a string: "*ComponentHomeFinder*"[OMG02] [OMG02b].

De posse da referência ao *Home*, o cliente executa uma das operações *factory* [GOF00][OMG02] e obtém a referência à interface equivalente do componente. A interface equivalente oferece os meios para que um cliente obtenha as referências às portas dos componentes e estabeleça as conexões entre elas.

A obtenção da referência à interface equivalente do componente é representada na figura 4-3:

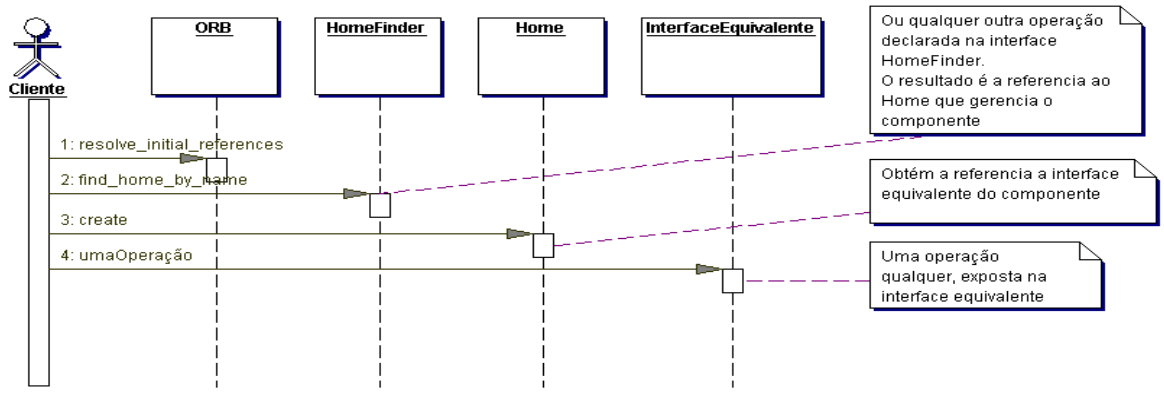

Figura 4-3 - Obtenção da interface equivalente de um componente à partir de um cliente.

No momento em que o cliente obtém a referência à interface equivalente do componente, no servidor, nenhum dos objetos CORBA que formam o componente foi instanciado. A instanciação acontece de fato no momento em que o objeto CORBA recebe a primeira requisição e o primeiro *preinvoke* é executado no *servant locator*.

No caso da interface equivalente do componente, além de criar uma instância do executor, o primeiro *preinvoke* causa também a criação de uma instância do executor do objeto contexto e do executor local do componente (objeto CORBA local que implementa as operações de *callback* do componente) e, em seguida a execução da operação de *callback set_session_context* (no executor local do componente) seguido do acionamento da operação *ccm_activate* (também de *callback*).

Como a implementação propriamente dita das operações de negócios deve ser realizada pelo desenvolvedor do componente e os executores locais que implementam as interfaces de *callback* já tem naturalmente seu código usado pelos desenvolvedores, o protótipo adota que todas as operações que são declaradas nas interfaces externas e que existem nos executores locais são efetivamente executadas no executor local.

Ou seja, todas as operações recebidas por um executor de interface externa que tenham uma operação com assinatura igual no executor local são delegadas para ele, evitando assim que o desenvolvedor tenha contato com o código gerado para as interfaces internas.

Um *servant* é instanciado (ou obtido do *pool*) no primeiro acionamento da operação *preinvoke* do *servant locator* e a sua remoção acontece quando não receber requisições por um determinado período de tempo ou quando for explicitamente solicitado pelo cliente.

O comportamento do *servant locator* quando a primeira operação destinada ao componente é executada pelo cliente é representada na figura 4-4, onde: *NomeComponenteImpl* é o executor do componente, *CCM_NomeComponenteImpl* é o nome do executor local do componente e *CCM_NomeComponente_Context* é o contexto do componente.

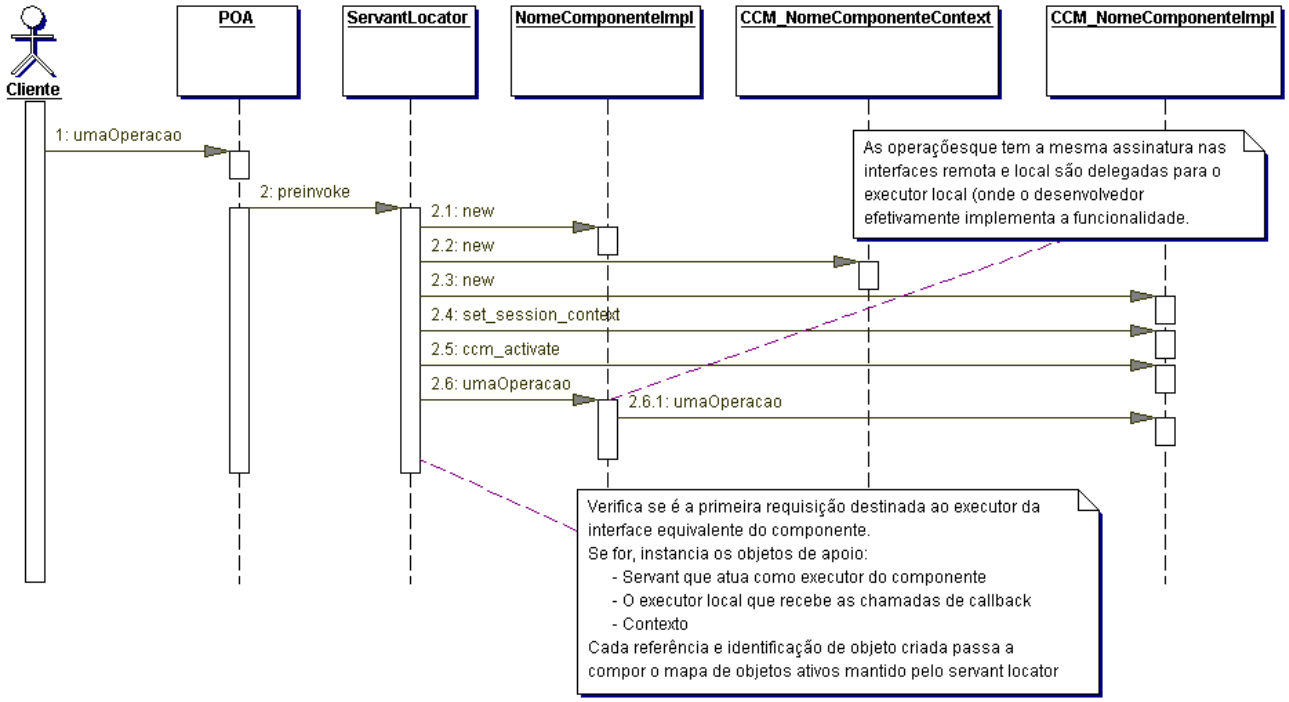

Figura 4-4 - *Servant locator* ao receber a primeira operação na interface equivalente do componente.

4.1.1 IMPLEMENTAÇÂO DO OBJETO CONTEXTO

Cada componente tem uma interface contexto declarada durante a conversão de IDL estendida para IDL equivalente, cujo nome respeita a seguinte regra de formação: *CCM_<nomeDoComponete>_Context*.

A nova interface especializa a interface *SessionContext* se o componente for básico e de *Session2Context* se o componente for estendido. *Session2Context* é também uma especialização de *SessionContext* e ambas fazem parte da API *session*. *SessionContext* e *Session2Context* são respectivamente, especializações das interfaces: *CCMContext* e

CCM2Context. A figura 4-5 ilustra a hierarquia existente entre as interfaces internas do modelo de componentes CORBA para a API *session*:

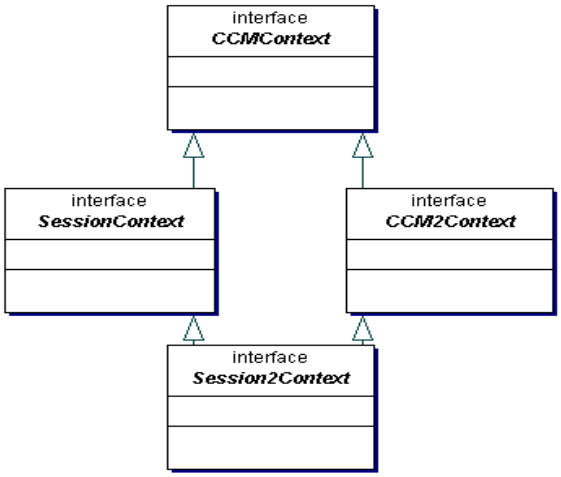

Figura 4-5 - Hierarquia entre interfaces internas que especializam *CCMContext* para a API de *sessão*.

Um arquivo IDL equivalente contendo uma declaração de contexto conforme citado, foi reproduzido no anexo 2 deste trabalho.

A implementação do contexto de cada componente é realizada pelo container. Cada *servant* instanciado para o objeto contexto oferece as operações declaradas nas interfaces herdadas. A listagem 4-1 mostra a declaração da IDL da interface *CCMContext*, reproduzida de [OMG02]:

```
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);
};
```

Listagem 4-1- Declaração IDL da interface *CCMContext*.

Para a operação *get_CCM_Home* obter a referência ao *home*, usa-se a operação *getCCMHomeUsingHomeName* da classe *DeployedHomesSingleton,* as demais operações dizem respeito ao uso do serviço de transações, que não está disponível no protótipo.

A interface *SessionContext* expõe a operação *get_CCM_Object* que, ao ser acionada, retorna a referência à interface equivalente do componente.

De acordo com a especificação, o acionamento desta operação deve ser realizado somente no escopo de uma operação de *callback* (em outras palavras, pode ser acionada somente a partir do executor local do componente, através de uma das operações herdadas

da interface *SessionComponent*). O protótipo não oferece implementação para esta operação.

A interface *CCM2Context* declara três operações, mas nenhuma é suportada pelo protótipo, a operação *get_home_registration* não é implementada, devido à solução adotada no protótipo para o registro de *Homes*. No protótipo, o registro dos *homes* acontece automaticamente sem usar a interface *HomeRegistration*, quanto a operação *get_persistence*, ela não se aplica ao protótipo dado que o contêiner não dá suporte ai serviço de persistência (desnecessário a componentes de sessão). A declaração IDL da interface *CCM2Context* é reproduzida na listagem 4-2 [OMG02]:

```
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);
};
```

Listagem 4-2 - Declaração IDL da interface CCM2Context.

O conjunto final de operações é herdado da interface *Session2Context*. De acordo com a especificação, esta interface "acrescenta a habilidade de criar referências aos componentes instalados em um contêiner para componentes de sessão", mas sem acionar o *Home* para isso.

Desta forma, a responsabilidade pelo gerenciamento das referências e identificações de objetos obtidos através destas operações é do implementador do componente (na interface de *callback*) e não do contêiner.

No protótipo, estas operações foram implementadas de forma que, quando acionadas, obtém o POA responsável pelo gerenciamento do componente identificado pelo parâmetro passado (mesmo no caso de se informar uma porta, o POA do componente é localizado). A execução das operações *create_ref* e *create_ref_from_id* executam a operação *create_ref_with_id* do POA obtido.

A operação *get_oid_from_ref* obtém a identificação do objeto da operação *reference_to_id* também do POA localizado.

A declaração em IDL da interface *Session2Context* é reproduzida na listagem 4-3 [OMG02]:

```
enum BadComponentReferenceReason {
       NON_LOCAL_REFERENCE, NON_COMPONENT_REFERENCE, WRONG_CONTÊINER
};
exception BadComponentReference {
       BadComponentReferenceReason reason;
};
exception IllegalState { };
local interface Session2Context : SessionContext, CCM2Context {
       Object create_ref (in CORBA::RepositoryId repid);
       Object create_ref_from_oid (
                           in PortableServer::ObjectIdCORBA::OctetSeq oid,
                           in CORBA::RepositoryId repid);
       PortableServer::ObjectId
       CORBA::OctetSeq get_oid_from_ref (in Object objref)
                  raises (IllegalState, BadComponentReference);

};
```

Listagem 4-3 - Declaração IDL da interface Session2Context.

Um pormenor importante sobre o funcionamento do componente em relação ao modelo é que, no modelo exposto na especificação, os objetos CORBA que implementam as interfaces internas e externas de um componente não têm acesso ao contexto, somente as interfaces de *callback* o tem.

Mas a conexão entre portas é estabelecida através das operações expostas nas interfaces externas que não têm acesso ao contexto. Por sua vez, o contexto é o elo existente entre a interface externa e as interfaces de *callback* (haja visto que as operações que são declaradas em IDL equivalente para dar suporte às portas na interface externa, têm a mesma assinatura das declaradas no contexto).

Para garantir que o estabelecimento ou o cancelamento de uma conexão entre facetas e receptáculos seja sinalizado para o contexto, no protótipo, a declaração da IDL equivalente do contexto inclui uma operação *set_connection_NomeDaFaceta* para cada faceta declarada. Além disso, o contexto mantém as referências às facetas e às portas emissoras de eventos como variáveis de instância para que possam ser devolvidas pelas operações usadas pelos objetos que implementam as interfaces de *callback*.

As operações "*set_connection_NomeDaFaceta*" são acionadas pelos objetos que implementam as interfaces externas sempre que uma conexão é estabelecida (ou desfeita) entre uma faceta e um receptáculo, atualizando a situação da conexão para o contexto.

Uma implementação do objeto contexto é reproduzida no anexo 3 deste trabalho.

4.1.2 IMPLEMENTAÇÂO DO OBJETO HOME

Os objetos *Home* dos componentes *session*, implementam as operações das interfaces *CCMHome* e *KeylessCCMHome* além das operações declaradas em IDL (próprias do componente).

CCMHome e *KeylessCCMHome* são herdadas respectivamente pelas interfaces explícitas (*Explicit*) e implícitas (*Implicit)* que por sua vez são especializadas na interface *home* do componente. A figura 4-6 é um diagrama que representa a hierarquia existente entre as interfaces citadas:

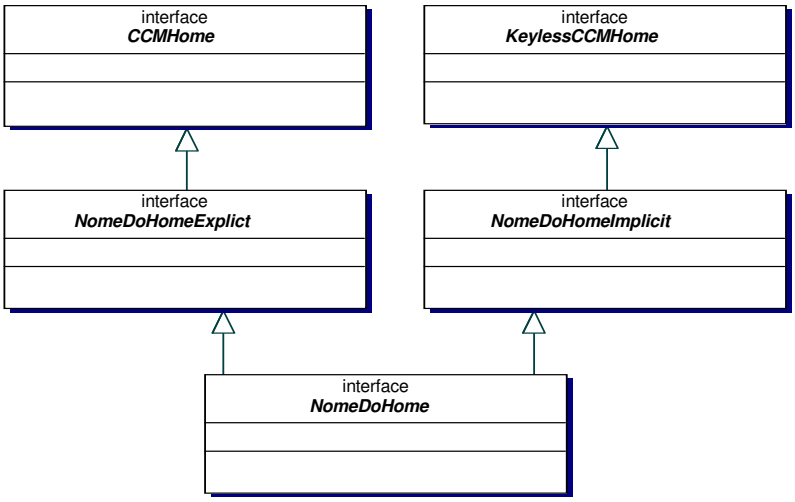

Figura 4-6 - Hierarquia entre interfaces externas do tipo Home.

O protótipo implementa as operações herdadas da interface *CCMHome*: *get_component_def* e *get_home_def* cujos retornos são respectivamente objetos do tipo: *CORBA::ComponentIR::ComponentDef e CORBA::ComponentIR::HometDef*.

A interface *ComponentDef* [OMG02b] contém a representação completa da IDL do componente, com listas mantendo os valores das declarações: *provides*, *uses, consumes*, *publishes*, *supports*, *emits* e os atributos.

A interface *HomeDef* [OMG02b] é análoga à *ComponentDef* e contém a representação completa da IDL do *home*, com listas mantendo os valores das declarações: constantes, tipos, *exceptions*, operações, atributos, *fatories* e *finders*.

No caso do protótipo, os *servants* dos objetos que implementam estas interfaces contém somente as operações que atualizam o atributo "id" (leitora – *getter* e escritora – *setter*). As demais operações não são usadas pelo protótipo e, portanto não são implementadas.

Estes objetos são criados durante a inicialização do contêiner e armazenado no *singleton*: *ComponentDefListSingleton*.

No protótipo, a operação "*remove_component"* aciona a operação "*remove*" do componente passado como parâmetro. O que é curioso sobre esta operação é que qualquer

componente pode ser “removido” à partir de qualquer *Home* , dado que a especificação não dita que é preciso consistir o tipo de componente recebido como parâmetro

A declaração da interface *CCMHome* em IDL é reproduzida na listagem 4-4 [OMG02]:

```
typedef unsigned long FailureReason;
exception RemoveFailure { FailureReason reason; };
interface CCMHome {
       CORBA::IRObject get_component_def();
       CORBA::IRObject get_home_def ();
       void remove_component ( in CCMObject comp
            raises (RemoveFailure);

};
```

Listagem 4-4 - Declaração da interface *CCMHome*

4.1.3 IMPLEMENTAÇÃO DOS EXECUTORES DOS COMPONENTES

Os componentes CCM estendidos podem ter portas e, para permitir a navegação entre elas de forma introspectiva, o componente deve implementar operações comuns que estão declaradas em interfaces herdadas pela interface *CCMObject* e consequentemente também pelas interfaces equivalentes dos componentes (todo o componente especializa a interface *CCMObject*).

Todo o *servant* que atua como executor de componente deve especializar a interface *CCMObject* que por sua vez especializa as interfaces *Navigaton*, *Receptacles*, *Events*, como demonstrado na figura 4-7:

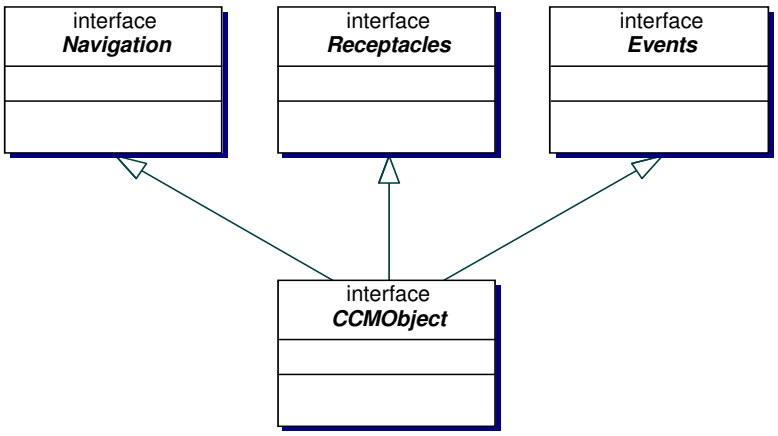

Figura 4-7 - Interfaces herdadas por *CCMObject*.

Das operações herdadas de *CCMObject*, os componentes executados no protótipo implementam as operações: *get_component_def* e *get_ccm_home*.

A declaração IDL da interface *CCMObject* está reproduzida na listagem 4-5 [OMG02]:

```
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 ();
};
```

Listagem 4-5 - Declaração da interface *CCMObject*

Além das operações da interface CCMObject somente a operação "*same_component*", da interface *Navigation*, foi implementada. Esta operação verifica se a referência passada como parâmetro corresponde a um objeto interno da mesma "instância" do componente sendo executado. Foi implementada para demonstrar que o protótipo garante a integridade de cada instância do componente, mantendo registro de todos os objetos que formam uma determinada instância de um componente.

Nenhuma operação da interface *Events* foi implementada porquê as funcionalidades pedidas pelas operações nela declaradas já são atendidas pelas operações geradas em IDL estendida (obtenção de portas, conexão, etc ...)

4.1.4 PORTAS EMISSORAS E RECEPTORAS DE EVENTOS

No protótipo a troca de mensagens assíncronas (realizada através das portas receptoras e emissoras de eventos) é feita usando-se o serviço de notificações CORBA.

Em uma aplicação CORBA tradicional que usa o serviço de notificações, identificam-se especialmente duas entidades [OMG02a] [BOL02]:

- **O provedor ou *supplier*** – É o emissor de eventos propriamente dito. Trata-se de um objeto CORBA que, no modo de operação *push* (que é o usado pelo *contêiner*), especializa a interface *PushSupplier* (ou uma de suas derivadas).
- **O consumidor ou *consumer*** – Cada um dos receptores de eventos que assinam a ao menos um canal de eventos é um consumidor. Trata-se de um objeto CORBA que, no modo de operação *push*, especializa a interface *PushConsumer* (ou uma de suas derivadas).

A interação entre o *supplier* e o *consumer* depende de uma "conexão":

- **No lado *consumer*** é a assinatura ou divulgação do interesse em receber eventos do canal de eventos e
- **No lado *supplier*** é o registro do interesse em publicar eventos no canal de eventos.

Somente depois de estabelecidas as conexões é que os *suppliers* podem publicar eventos e os *consumers* podem recebê-los. As conexões são feitas nos canais de eventos por

meio de *proxies* (objetos CORBA que mediam a comunicação entre *supplier* e *consumer*) [GAM00] [OMG02a].

Se for abordado de forma bastante simplificada, o processo de envio e recepção de eventos usando o modo de operação *push* [OMG02a], consiste de fazer com que um *supplier* execute a operação *push* (passando como parâmetro os dados do evento) em cada um dos *consumers* assinantes na ocorrência de um evento [BOL02].

Para conseguir que esta tarefa seja realizada de forma assíncrona existe a figura do canal de eventos (ou, no caso do serviço de notificações, "canal de notificações", mas para os propósitos deste trabalho: canal de eventos e canal de notificações serão usados como sinônimos) e para permitir que o *supplier* não precise conhecer cada um dos *consumers* existe a figura do *proxy*.

Um *supplier* interage com um *proxyConsumer* e um *consumer* interage com um *ProxySupplier*. Os *proxies* interagem com o canal de eventos e com os respectivos *suppliers* e *consumers* que os assinaram, como representado na figura 4-8:

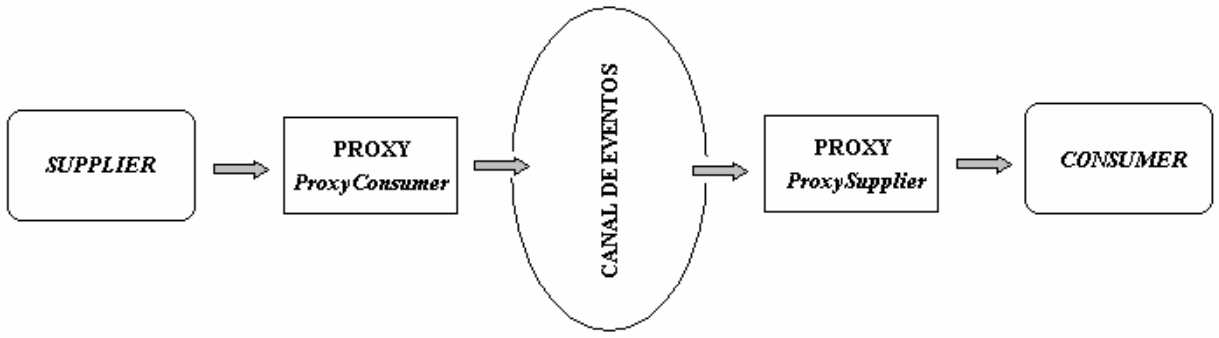

Figura 4-8 - Interação entre os envolvidos no processo de envio de eventos

De posse das referências aos objetos *proxy*, tanto os *suppliers* quanto os *consumers* podem se conectar ao canal de eventos (a conexão consiste de executar uma operação passando a referência ao objeto *supplier* ou *consumer* para que seja usada pelo canal de eventos).

Assim que o canal é criado, o *supplier* obtém do canal uma referência a um objeto *SupplierAdmin* através da operação *for_suppliers* para então solicitar o *proxy* que será usado no envio dos eventos. De posse do *proxy*, basta ao *supplier* solicitar a conexão (através da operação *connect_push_supplier*).

O diagrama apresentado na figura 4-8 representa a assinatura e o envio de um evento realizado por um *supplier*.

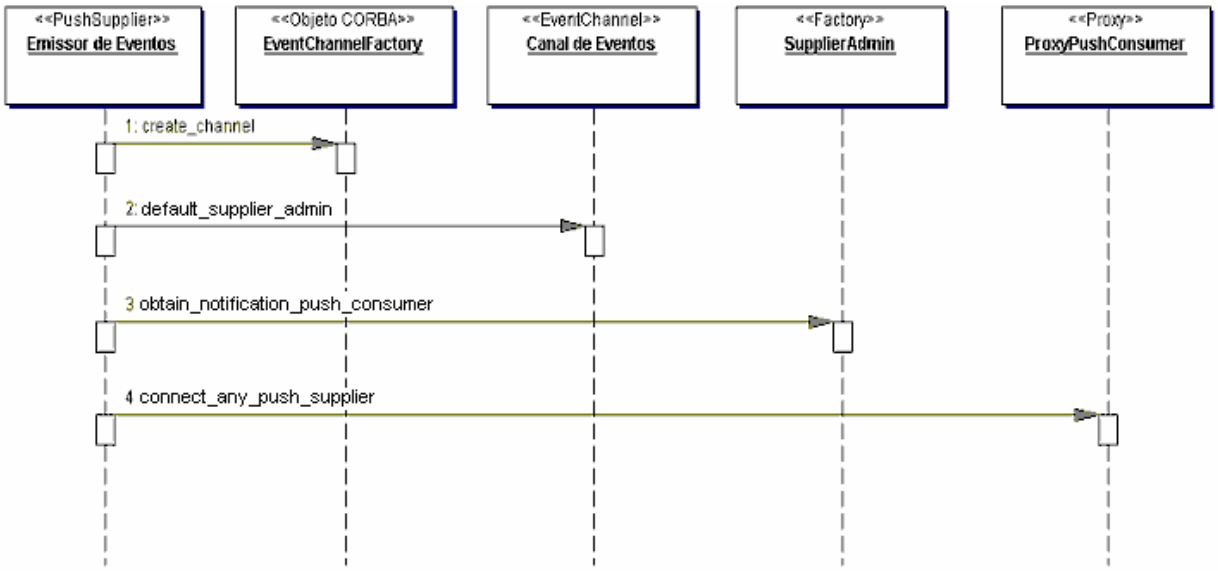

Figura 4-9 - Obtenção de um *proxy* e assinatura de um canal de eventos a partir de um *supplier*.

Uma vez conectado ao canal de eventos por meio do *proxy*, o *supplier* é capaz de publicar seus eventos no canal de eventos. Isso acontece acionando-se a operação *push* do *proxy*, como representado na figura 4-9:

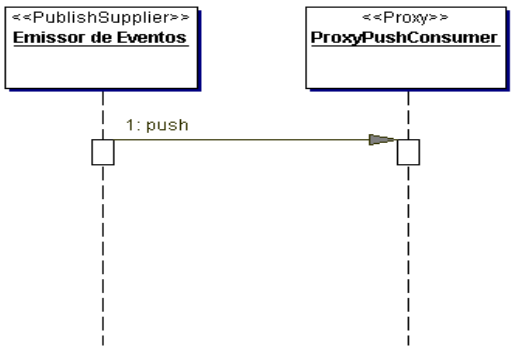

Figura 4-10 - Publicação de um evento no canal de eventos

Para o *consumer*, o processo de conexão é semelhante ao descrito na figura 4-8, a diferença está nos nomes das operações e nos tipos de objetos usados:

- A obtenção do *factory* é conseguida através da operação *default_consumer_admin* em lugar de *default_supplier_admin*;
- O *factory* obtido é do tipo *ConsumerAdmin* em lugar de *SupplierAdmin*;
- A obtenção do *proxy* acontece através da operação *obtain_notification_push_supplier* em lugar de *obtain_notification_push_consumer*;
- O *proxy* obtido é do tipo *ProxyPushSupplier* em lugar de *ProxyPushConsumer*

- A operação que realiza a conexão é *connect_any_push_consumer* em lugar de *connect_any_push_supplier*

Um cuidado importante a ser tomado é garantir que o canal de eventos sendo assinado seja o mesmo, tanto para o *supplier* quanto para o *consumer*.

4.1.5 MAPEAMENTO ENTRE O SERVIÇO DE NOTIFICAÇÕES CORBA E O MODELO CCM NO PROTÓTIPO

Em uma aplicação CCM, a figura dos participantes da troca de mensagens assim como a criação e uso do canal de eventos são isolados do desenvolvedor através das portas emissoras e receptoras de eventos.

Para o desenvolvedor do componente, a troca de mensagens entre os componentes CCM é trivial, principalmente porque as interfaces e boa parte da implementação é gerada pelos utilitários de instalação, mas para o implementador do contêiner o caso é diferente.

A especificação é generosa quanto a detalhes ligados à implementação dos componentes, mas é dúbia quanto as funcionalidades que as interfaces que devem ser declaradas durante a compilação de IDL estendida para IDL equivalente devem efetivamente realizar.

Para CCM, todos os eventos devem ser declarados pelo desenvolvedor de componentes em IDL estendida como um "tipo de evento". A listagem 4-6 representa a declaração de um evento em IDL estendida:

```
eventtype NomeEvento {
       // Declaração dos atributos que identificam uma instância do tipo de Evento
};
```

Listagem 4-6 - Declaração de um evento em IDL estendida

Quando convertido para IDL equivalente, o tipo de evento se torna um *valuetype*, um tipo de dado especial que permite a uma operação passa-lo como parâmetro a uma outra operação "por valor".

Embora seja declarado em IDL e gerado pelo compilador IDL, não se trata de um objeto CORBA (não é associado a um ORB, não tem identificação de objeto e não é acionado por um POA).

A operação que recebe o *valuetype* usa uma cópia local do objeto e não uma referência a um objeto executado no servidor. Os *valuetypes* que declaram eventos são do tipo *EventBase*, eles contém um conjunto de atributos que o caracterizam e identificam o estado de um evento.

Para cada declaração de tipo de evento, o modelo declara também uma interface que deve ser implementada como um executor local para a porta, a interface é nomeada como "CCM_" + nome da interface original + "*Consumer*".

A declaração apresentada na listagem 4-6 convertida para IDL equivalente é representada na listagem 4-7 [OMG02].

```
valuetype NomeEvento:Components::EventBase{
       // atributos
};
interface NomeEventoConsumer:Componentes::EventConsumerBase {
       void push_NomeEvento (in NomeEvento tipoDeEvento);
};
local interface CCM_NomeEventoConsumer {
       void push (in NomeEvento tipoDeEvento);
};
```

Listagem 4-7 - Declaração das interfaces geradas à partir do evento descrito em IDL estendida

A única informação existente na especificação sobre a interface do executor local, nomeada na listagem 4-7 como "*CCM_NomeEventoConsumer*" é a que está reproduzida a seguir: "Para cada *eventype*, uma interface para o executor local do consumidor deve ser declarada".

Não existem indicações sobre quando a operação *push* nela declarada deve ser acionada ou qual a sua função para o modelo CCM. Se for mantida como definido, trata-se de uma interface desnecessária que poderia ser removida da especificação sem maiores danos ao funcionamento do modelo de componentes.

Durante a compilação de IDL estendida para IDL equivalente, tanto o componente que tem portas emissoras de eventos quanto os componentes que tem portas receptoras de eventos recebe uma interface de nome <*NomeDoEvento*>*Consumer* declarada no módulo <NomeDoComponente>*EventConsumers*.

A listagem 4-8 representa as interfaces geradas, supondo dois componentes que trocam o evento chamado "NomeEvento", o primeiro chamado de "NomeComponenteEmissor" e o segundo "NomeComponenteReceptor" e que eles expõem respectivamente as portas: "NomeEventSource" e "NomeEventSink":

```
module NomeComponenteEmissorEventConsumers {
       interface NomeEventSourceConsumer:Components::EventConsumerBase {
              void push (in NomeEvento evt);
       };
};

module NomeComponenteReceptorEventConsumers {
       interface NomeComponenteReceptorConsumer:Components::EventConsumerBase {
              void push (in NomeEvento evt);
       };
};
```

Listagem 4-8 - Interfaces "*EventConsumers*" criadas para componentes com portas que trocam eventos.

Assim como acontece com a declaração da interface "*CCM_NomeEventoConsumer*" (listagem 4-7), o papel destas interfaces "*EventConsumers*" não é claro, embora a especificação permita inferir que elas assumam o papel de *proxies* entre o modelo de troca de eventos do modelo de componentes e o serviço usado pelo contêiner (no caso do protótipo, o serviço de notificações, mas poderia ser qualquer outro serviço que oferecesse a mesma funcionalidade).

Esta nomenclatura causa bastante confusão durante a codificação, visando melhorar a semântica destas interfaces, a interface que é declarada para o componente que expõe a porta emissora de eventos deveria receber sufixo "*Supplier*" em lugar de "*Consumer*" (baseado na nomenclatura usada no serviço de notificações CORBA [OMG02a]), deixando claro o seu papel para o desenvolvedor.

Além das interfaces citadas, são declaradas também operações para o objeto contexto e para o executor do componente que expõe portas emissoras de eventos. A declaração destas operações é outro ponto ambíguo na especificação.

No texto [OMG02] é escrito que: "A operação *subscribe_<source_name>* conecta o consumidor passado como parâmetro a um canal de eventos oferecido à implementação do componente pelo contêiner", mas o parâmetro recebido pela operação é o *consumer* do próprio componente, ou seja, o *supplier*.

Além disso, a operação s*ubscribe_nomeEventSource* tem duas versões declaradas, uma recebendo *NomeEventoConsumer* e outra recebendo *NomeComponenteEventConsumers::NomeEventoConsumer*. O mesmo acontece com o retorno da operação *unsubscribe_nomeEventSource* que também é declarada em duas versões.

A listagem 4-9 representa as operações declaradas em IDL para um componente que oferece uma porta emissora de eventos:

```
interface NomeComponenteEmissor:Components::CCMObject {
       Components::Cookie subscribe_nomeEventSource (
              in NomeComponenteEmissorEventConsumers::NomeEventoConsumer consumer
              ) raises ( Components::ExceededConnectionLimit );

       NomeComponenteEmissorEventConsumers::NomeEventoConsumer
              unsubscribe_nomeEventSource ( in Components::Cookie ck
              ) raises ( Components::InvalidConnection );
};

local interface CCM_NomeComponenteEmissor_Context: Components::Session2Context{
       void push_nomeEventSource(in NomeEvento ev);
};
interface NomeComponenteEmissor : Components::CCMObject {
       Components::Cookie subscribe_nomeEventSource( in NomeEventoConsumer consumer
              ) raises (Components::ExceededConnectionLimit);

       NomeEventoConsumer unsubscribe_nomeEventSource( in Components::Cookie ck
              ) raises (Components::InvalidConnection);
};
```

Listagem 4-9 - Interfaces geradas para o componente que expõem portas emissoras de eventos.

O componente que expõe a porta consumidora de eventos também recebe duas versões da operação: *get_consumer_NomeEventSink*, um que retorna um objeto CORBA do tipo: *NomeComponenteReceptorConsumer* e outra que retorna.*NomeComponenteReceptorEventConsumers:: NomeComponenteReceptorConsumer* e uma operação de *callback* no executor local .

A operação declarada no executor local do componente que tem uma porta receptora de eventos é a que efetivamente recebe o evento do contêiner. A listagem 4-10 representa a interfaces geradas:

```
interface NomeComponenteReceptor:Components::CCMObject {
       NomeComponenteReceptorEventConsumers::NomeEventoConsumer
       get_consumer_nomeEventSink ();
};

interface CCM_NomeComponenteReceptor:Components::SessionComponent{
       void push_nomeEventSink (in NomeEvento evento);
};

interface NomeComponenteReceptor:Components::CCMObject {
       NomeEventoConsumer get_consumer_nomeEventSink ();
};
```

Listagem 4-10 - Operações declaradas o componente que expõe portas receptoras de eventos.

Visando melhorar a semântica da troca de eventos no modelo CCM, as interfaces declaradas nos módulos "*NomeComponentEventConsumers*" (como na listagem 4-9), passam a especializar a interface declarada para o evento como *NomeEventoConsumer* (declarado na listagem 4-8).

Desta forma, em lugar de se ter duas versões para as operações: *subscribe* e *unsubscribe* no componente que expõe uma porta emissora de eventos (como representado na listagem 4-9) e para a operação *get_consumer* para o componente que expõem uma porta receptora de eventos (como representado na listagem 4-10), a versão estendida da IDL passa a declarar somente as operações que recebem como parâmetros ou que retornam como resultado a interface *NomeEventoConsumer*.

Além disso, as interfaces criadas para cada componente no pacote *NomeComponenteEventConsumers* passam a assumir o seu papel de *consumer* ou *supplier* no serviço de notificações, especializando *CosEventComm::PushConsumer* ou *CosEventComm::PushSupplier*.

A figura 4-11 ilustra a hierarquia das interfaces usadas para uma porta emissora de eventos. Uma porta receptora de eventos tem a mesma hierarquia, mas em lugar de especializar *PushSupplier*, especializa *PushConsumer*.

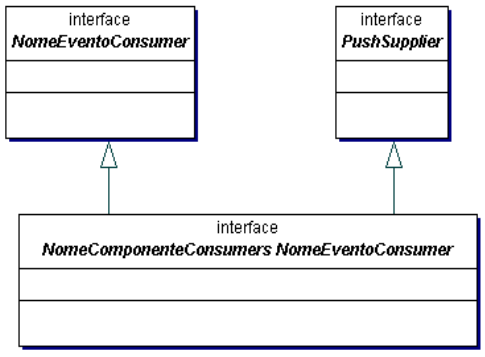

Figura 4-11 - Hierarquia de interfaces para portas emissoras de eventos como proposto no protótipo.

Com a hierarquia adotada no protótipo para as portas emissoras e receptoras de eventos, as interfaces "*Consumer*" declaradas na especificação, passam a assumir explicitamente o papel de "*Supplier*" no componente que tem a porta emissora de eventos e de "*Consumer*" no componente que tem a porta receptora de eventos.

Quanto à implementação do componente, para que um evento seja "lançado" no lado "emissor de eventos", o desenvolvedor deve codificar no executor local do componente uma chamada à operação *push_nomeDoEvento* do objeto contexto. Esta operação executa a operação *push* no *consumer* do componente (que atua como *supplier* junto ao serviço de notificações conforme representado na figura 4-12).

O serviço de notificações se encarrega de acionar a operação *push* do *consumer* do componente consumidor que foi conectado a porta emissora. Este executa o *push* no executor local do componente, operação esta que deve ser codificada pelo implementador do componente.

A figura 4-12 ilustra a sequência de atividades envolvidas com o envio de eventos entre uma porta emissora e uma porta receptora de eventos:

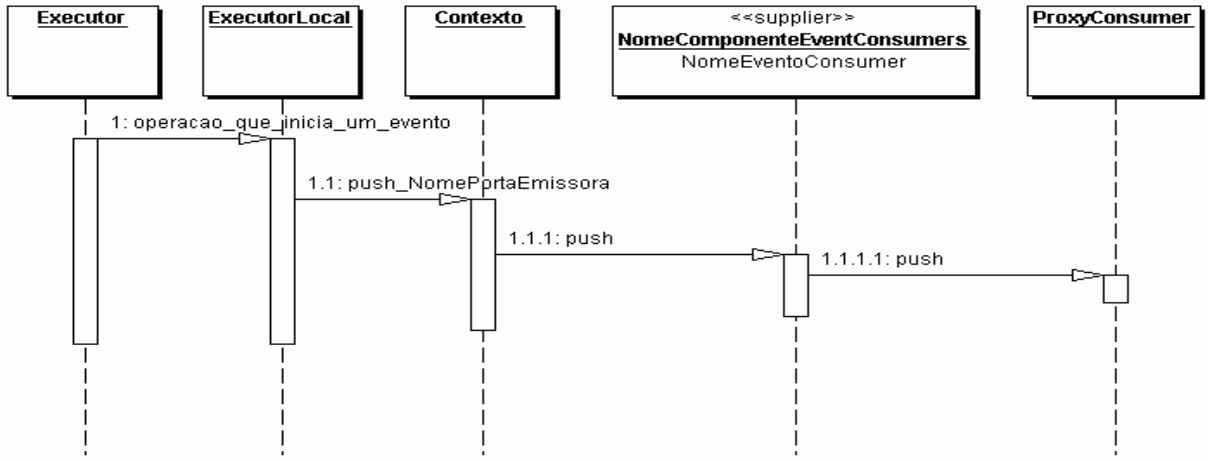

Figura 4-12 - Sequência de envio de um evento.

Antes que o envio de eventos possa ocorrer, uma conexão entre as portas consumidoras e emissoras de eventos deve ser estabelecida, para tanto, um cliente deve obter a referência à porta consumidora usando-se a operação *get_consumer_NomeDaPortaReceptora* e passa-la como parâmetro para a operação *subscribe_NomeDaPortaEmissora* no componente que tem a porta emissora.

No protótipo, a operação *get_consumer_NomeDaPortaReceptora* é responsável por obter a referência e a identificação do objeto para a porta receptora de eventos, que é o *consumer* para o serviço de notificações do contêiner.

A operação *subscribe_NomeDaPortaEmissora* é responsável pela criação da referência e identificação do objeto para a porta emissora de eventos (o *supplier*), além disso inicializa toda a ligação das portas com o serviço de notificações: cria o canal de eventos, obtém os proxies e os conecta ao *supplier* e ao *consumer*.

A especificação indica que o objeto contexto deve oferecer a operação *push_NomeDaPortaEmissoraDeEventos* que deve ser usada pelo componente para efetivamente enviar o evento para as portas que "assinaram" o evento oferecido pela porta emissora. No entanto, o objeto que assume o papel de *supplier* no protótipo é o "*consumer*" criado para o componente e para que o contexto tenha a referência a este objeto foi incluída uma nova operação, chamada: *set_supplier_NomeDaPortaEmissora* que recebe como parâmetro a referência ao *supplier*. Ela deve ser acionada no executor do componente sempre que uma conexão for estabelecida ou desfeita.

De forma semelhante, para que o *supplier* possa efetivamente acionar a operação *push* no *proxy* correto, este precisa ter a referência ao *proxy*. Como o *proxy* é criado no executor do componente, no protótipo, a interface do *supplier* recebe uma operação extra (não prevista na especificação) chamada: *setProxyConsumer* que recebe a referência ao *proxy* quando a conexão entre as portas é estabelecida.

4.2 CONCLUSÕES

A conversão de IDL estendida para IDL equivalente deixa claro que cada componente CORBA é, na verdade, um conjunto de objetos CORBA e que o contêiner deve garantir a integridade do componente por eles formado.

Desta forma, o protótipo instancia um POA e um *servant manager* para cada tipo de objeto que faz parte do componente no momento da inicialização. Todos os POAs instanciados para os objetos do componente aplicam o mesmo conjunto de políticas.

O uso de *singletons* para manter a coesão entre os objetos CORBA, contendo as referências aos objetos, as respectivas identificações de objetos e as instâncias dos *servants* ligadas à identificação de cada instância de componente (interface equivalente) parece uma boa idéia, mas precisa de testes em ambientes com mais usuários executando operações simultâneas de remoção e obtenção de referências a objetos.

A alteração das interfaces "*Consumer"* geradas para as portas emissoras e receptoras de eventos para que especializassem o "*Consumer*" declarado para o *EventType*.também se provou uma boa opção de projeto, especialmente porque é totalmente compatível com qualquer componente que tivesse implementado os executores.

O contêiner implementado oferece um sub-conjunto de características que permite efetivamente avaliar o funcionamento interno de um componente CCM e sua interação com o contêiner.

5 UMA APLICAÇÂO DE EXEMPLO PARA USO NO PROTÓTIPO

O exemplo tem por objetivo demonstrar as características do modelo de componentes atendidas pelo protótipo.

Trata-se de uma implementação para o problema dos “filósofos glutões”. A opção por esta aplicação de exemplo foi feita levando-se em consideração o fato de que se trata de um problema clássico e conhecido pela grande maioria da comunidade acadêmica.

O objetivo do capítulo não é explorar as várias dimensões do problema dos filósofos glutões, mas sim, demonstrar o funcionamento de uma aplicação simples baseada em componentes sobre o protótipo e as características do modelo CCM suportadas pelo protótipo, por este motivo, a aplicação foi propositalmente modelada pra que todos os tipos de portas fossem usadas.

5.1 DESCRIÇÃO DA APLICAÇÃO DE EXEMPLO

O objetivo da aplicação é acompanhar e registrar as mudanças de estados sofridas por um conjunto de filósofos “virtualmente” sentados ao redor de uma mesa redonda na qual existe um suprimento infinito de macarrão. Cada filósofo tem um garfo a sua direita e um prato a sua frente, mas para que possa comer o macarrão, o filósofo deve obter os dois garfos próximos a si.

Caso consiga obter os garfos, o filósofo deve comer por um período de tempo e em seguida liberar os garfos, quando passa a pensar. Uma vez transcorrido o período em de tempo que esteve “pensando” o filósofo tenta novamente obter os garfos para comer. Caso não consiga, dorme. Quando acorda, tenta obter os garfos novamente.

Os períodos de tempo em que um filósofo fica pensando, comendo ou dormindo devem ser atribuídos no momento da criação de cada filósofo e para poder representar o uso de portas assíncronas disponíveis no protótipo, a cada mudança de estado, o Filósofo de “enviar” uma notificação para um observador (elemento que tem o papel de receber a alteração de estado de cada filósofo) .

Como o objetivo do exemplo é específicamente demonstrar características do protótipo, não é preciso que o aplicativo ofereça interação (interface com o usuário). Na verdade, assim que é acionado, o aplicativo deve criar filósofos, garfos e executar o gerenciamento de cada filósofo para que ofereçam o comportamento esperado.

A figura 5-1 é um diagrama de transição de estados que representa os estados que um filósofo pode assumir durante o seu ciclo de vida:

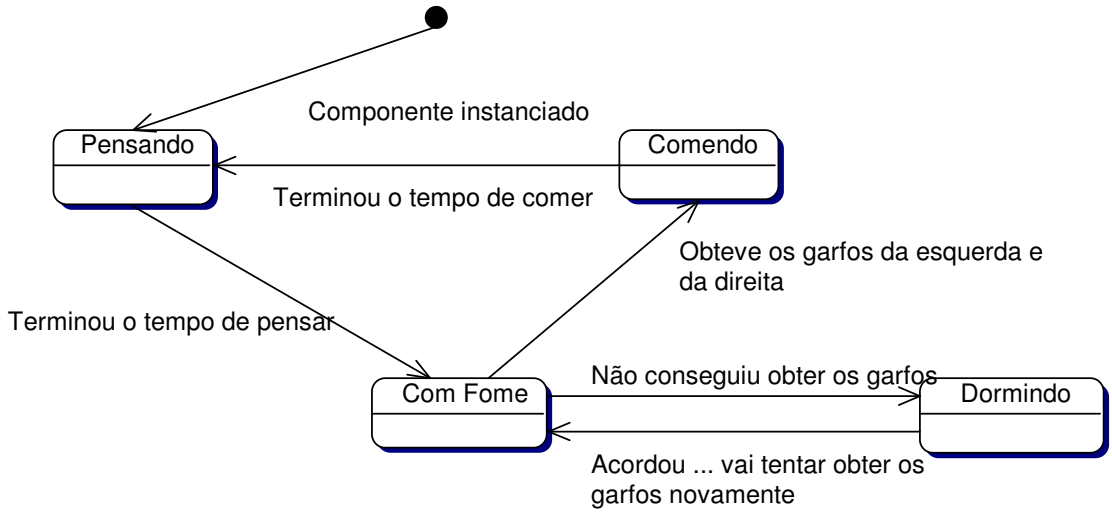

Figura 5-1 - Diagrama de Estados - Filósofos Glutões

A cada mudança de estado o filósofo informa a sua identificação e o seu novo estado a um observador.

5.1.1 DESCRIÇÃO DOS CASOS DE USO

Para melhor representar o conjunto de funcionalidades oferecido pelo exemplo, estas foram descritas na forma de casos de uso. A figura 5-2 ilustra os casos de uso contemplados no exemplo:

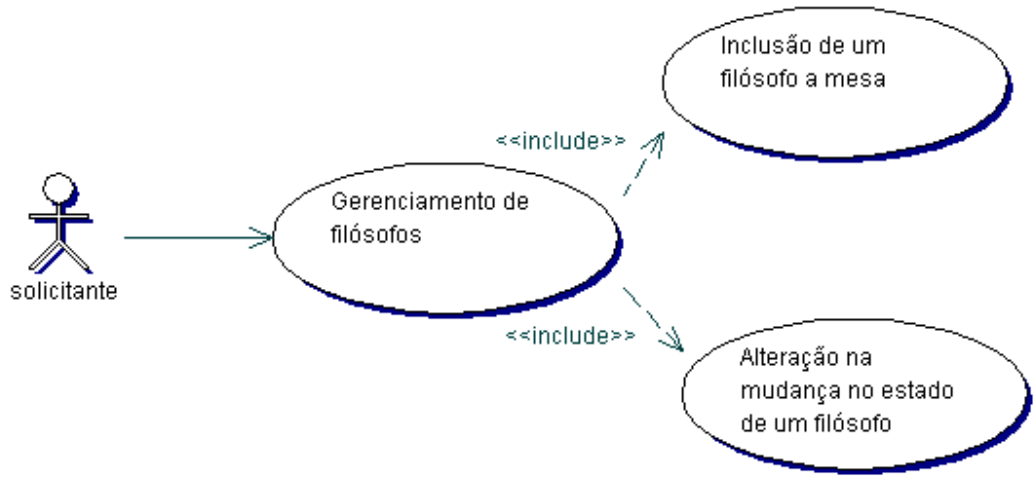

Figura 5-2 - Representação dos casos de uso contemplados no exemplo.

5.1.1.1 Caso de uso "Gerenciamento de filósofos"

Objetivos	Obter os filósofos e garfos e distribuí-los em volta da mesa, além disso, deve-se criar o observador e executar as tarefas de controle do ciclo de vida de cada filósofo (tempo em que fica comendo, pensando ou dormindo) e as ações que deve tomar a cada mudança de estado.
Atores	Solicitante – O usuário que pediu a execução do aplicativo.
Pré-condições	Os atributos que identificamnomes dos filósofos e os valores que identificam o tempo que levam para pensar, o tempo que leva para comer e o tempo que permanece dormindo devem estar definidos antes do início do caso de uso.
Inicialização	Acionado diretamente pelo "solicitante".
Fluxo principal	
1. O solicitante pede a execução do aplicativo	
2. O sistema obtém a lista de características dos filósofos que devem ser incluídos à mesa	
3. O sistema executa o caso de uso "inclusão de um filósofo a mesa" para cada filósofo	
4. O sistema "cria" um observador	
5. Para cada filósofo (simultaneamente), executa o caso de uso "alteração na mudança do estado de um filósofo".	
Pós-condição	Não se aplica.

5.1.1.2 Caso de uso "Inclusão de um filósofo a mesa"

Objetivos	"Criar" um filósofo e um garfo e incluir à mesa.
Atores	Não se aplica.
Pré-condições	Não se aplica.
Inicialização	Acionado pelo caso de uso "gerenciamento de filósofos"
Fluxo principal	
1. O sistema recebe as características de um filósofo (nome, identificação, tempo em que permanece pensando).	
2. O sistema cria um filósofo com as características pedidas;	
3. O sistema cria um garfo para o filósofo;	
4. O sistema inclui o filósofo e seu garfo à mesa.	
Pós- condição	O filósofo foi incluído à mesa e é "conectado" (passa a conhecer) o garfo a sua direita, o garfo a sua esquerda e o observador para quem deve informar as mudanças de estado.

5.1.1.3 Caso de uso "Alteração na mudança do estado de um filósofo"

Objetivos	Informar ao observador a ocorrência de uma mudança de estado.
Atores	Não se aplica.
Pré-condições	• O caso de uso "Inclusão de um filósofo à mesa" já foi executado para filósofo; • O caso de uso ""Gerenciamento de filósofos" já foi executado.
Inicialização	Acionado pelo caso de uso "gerenciamento de filósofos"
Fluxo principal	
1. O sistema aguarda o período de tempo declarado para o filósofo PENSAR;	
2. O sistema faz com que o filósofo notifique ao observador que está COM FOME;	
3. O sistema faz com que o filósofo tente obter os dois garfos (três tentativas);	
4. O sistema faz com que o filósofo notifique ao observador que está COMENDO;	
5. O sistema aguarda o período de tempo declarado para o filósofo comer;	
6. O sistema faz com que o filósofo "solte" os dois garfos;	
7. O sistema faz com que o filósofo notifique ao observador que está PENSANDO;	
Fluxo alternativo	
No passo 3 – Caso o filósofo não tenha conseguido obter os dois garfos (depois de três tentativas)	
1. O sistema faz com que o filósofo "solte" o garfo que eventualmente conseguiu obter.	
2. O sistema faz com que o filósofo notifique ao observador que está DORMINDO;	
3. O sistema aguarda o período de tempo declarado para o filósofo DORMIR;	
4. O sistema faz com que o filósofo notifique ao observador que está COM FOME;	
5. O sistema tenta executar o passo 3 (do fluxo principal) novamente	
Pós-condição	Não se aplica.

5.2 COMPONENTES USADOS NA REALIZAÇÃO DOS CASOS DE USO

Baseado na descrição do problema, a aplicação usada como exemplo foi modelada com três componentes: Filosofo, Garfo e Observador.

O componente "Filosofo" obtém ou libera "Garfos" por meio dos receptáculos: "maoEsquerda" e "maoDireita" que acessam as operações "pega" e "libera" do componente "Garfo" (operações oferecidas pela faceta "Garfo").

O Observador é informado sobre as alterações no estado do componente "Filosofo" por meio de uma porta "receptora de eventos" (*event sink*) que recebe um evento contendo: a identificação do filósofo (identificação e nome) e o novo estado deste. O envio destas informações é feito por meio de uma porta "emissora de eventos" (*event source*) do componente "Filosofo".

Todos os componentes são do tipo *session* porque precisam manter seu estado somente durante a execucao da aplição cliente. O componente: "Filosofo" tem os atributos: tempoComendo, tempoPensando, identificação e nome, que representam respectivamente: o tempo em que fica comendo (mantendo reservados para si os garfos da esquerda e da direita), tempo em que se mantém pensando, a identificação (número que representa o local que ocupa na mesa) e o seu nome.

A figura 5-3 representa a conexão entre as portas dos componentes modelados para o exemplo:

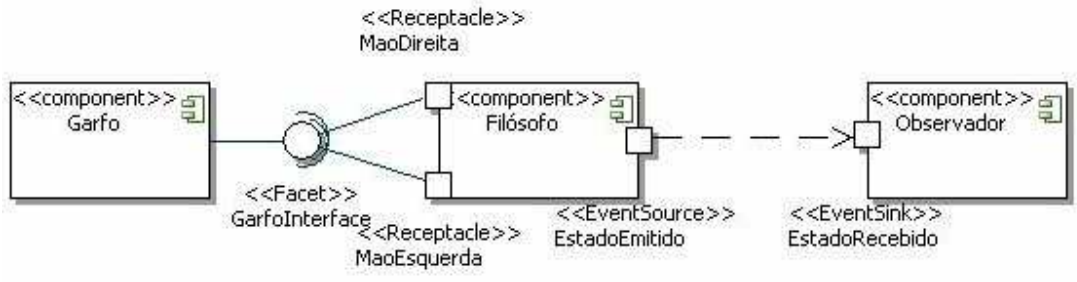

Figura 5-3 – Conexão entre as portas dos componentes criados para o exemplo.

5.3 APLICATIVO CLIENTE

Os componentes são acionados por um aplicativo cliente que atua como gerenciador de atividades para o componente. A aplicação cliente aciona as operações remotamente nos componentes CCM para:

1. Obter as referências aos *Homes* (usando o *HomeFinder*)
2. Obter as referências aos componentes filósofos, garfos e ao observador (operação *create* dos *Homes*)
3. Conectar as portas (liga os filósofos aos dois garfos a que tem acesso através dos recptáculos: "maoEsquerda" e "maoDireita" e ao observador através da porta emissora de eventos "estadoEmitido")

A figura 5-4 ilustra o funcionamento da aplicação cliente:

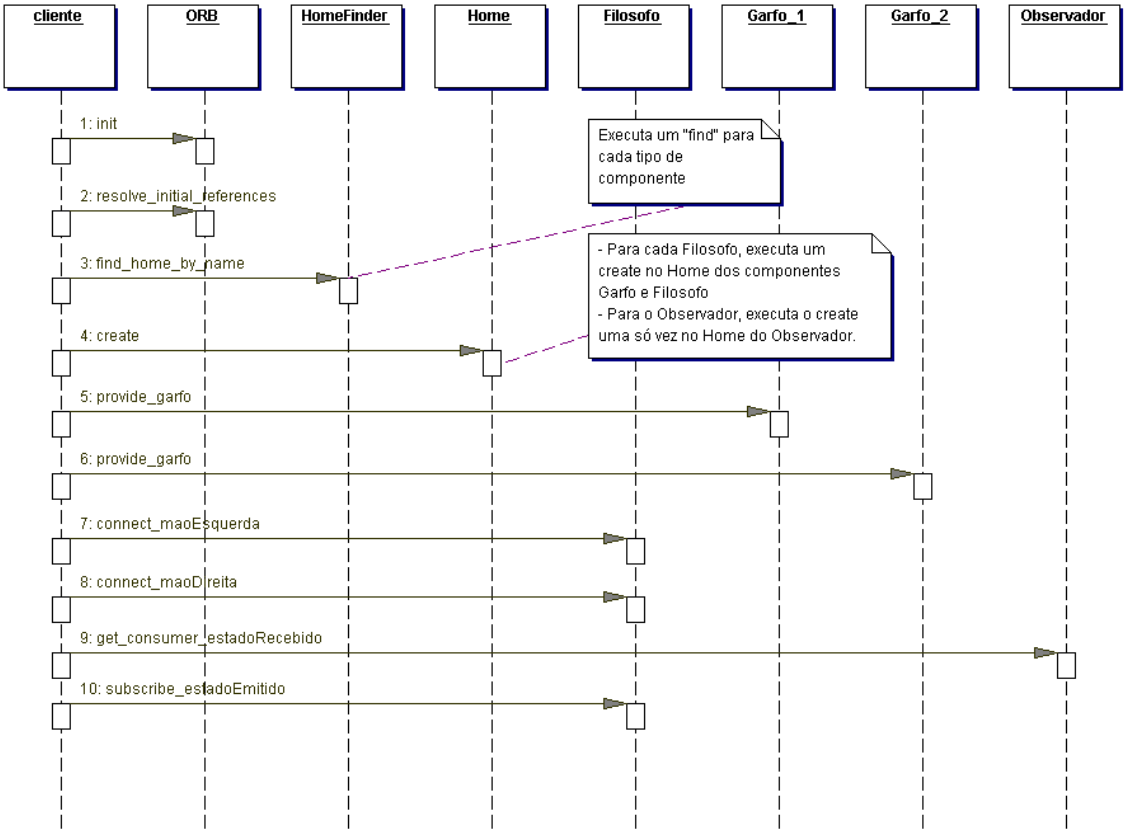

Figura 5-4 - Obtenção das referências aos componentes e estabelecimento das conexões etre portas.

Em seguida, uma *thread* é acionada (no cliente) para cada um dos filósofos. As *threads* são responsáveis pelo comportamento dos filósofos, que atuam como representado anteriormente na figura 5-1.

O controle da mudança de estados também é feito no cliente, que executa operações no componente filósofo que são responsáveis pelo envio de eventos: comendo, dormindo, comFome e pensando.

Somente as operações acessadas pelo cliente foram implementadas nos componentes O cliente não usa os recursos de introspecção e de navegação entre portas que normalmente estariam disponíveis através da interface equivalente do componente.

A confecção do cliente é a última parte do processo de criação de uma aplicação. O passo mais importante é a declaração das interfaces.

5.4 DECLARAÇÃO DAS INTERFACES E SUA IMPLEMENTAÇÃO

Um componente CCM deve ter as suas interfaces descritas em IDL estendida onde são declarados: os componentes propriamente ditos, as portas que usam ou oferecem, seus atributos e os *Homes* que gerenciam o componente, como listado no anexo 1.

Em IDL estendida, aplicando-se os critérios de conversão descritos na especificação CCM, um programa compilador deve gerar a interface equivalente, que por sua vez, é usada para a geração das interfaces na linguagem de programação escolhida através do compilador de IDL fornecido com o ORB, no caso do protótipo, na linguagem Java.

Para que a compilação seja possível, o arquivo IDL contendo a descrição das interfaces equivalentes deve fazer referência às declarações IDL para as interfaces do modelo CCM e para as interfaces IDL usadas pelos serviços CORBA, como o serviço de notificações [OMG02a], como listado no anexo 2.

Para o protótipo, a conversão de IDL estendida para IDL equivalente foi realizada manualmente, seguindo-se as instruções da especificação [OMG02].

Segundo estas instruções, no componente Garfo, a interface *CCM_Garfo* recebe a operação *get_garfo* que deve retornar a referência à interface local da faceta *CCM_GarfoInterface*.

A utilidade desta operação para um componente é questionável, dado que *get_garfo* não é acionada por *callback* e o executor local do componente é implementado pelo próprio desenvolvedor do componente, por isso, a obtenção da referência a esta interface seria realizada manualmente, sem acesso ao POA que gerencia os objetos CORBA que formam os componentes.

Para realmente fazer sentido, esta operação deveria estar disponível no contexto (*CCM_Garfo_Context*), assim, o executor local poderia ter acesso às referências às facetas do componente de forma controlada pelo contêiner.

A faceta propriamente dita é representada pela interface *GarfoInterface* e a referência ao objeto CORBA que implementa a interface é obtida usando-se a operação *provide_garfo* no objeto *Garfo*. A versão equivalente da declaração do componente Garfo é representada na figura 5-5:

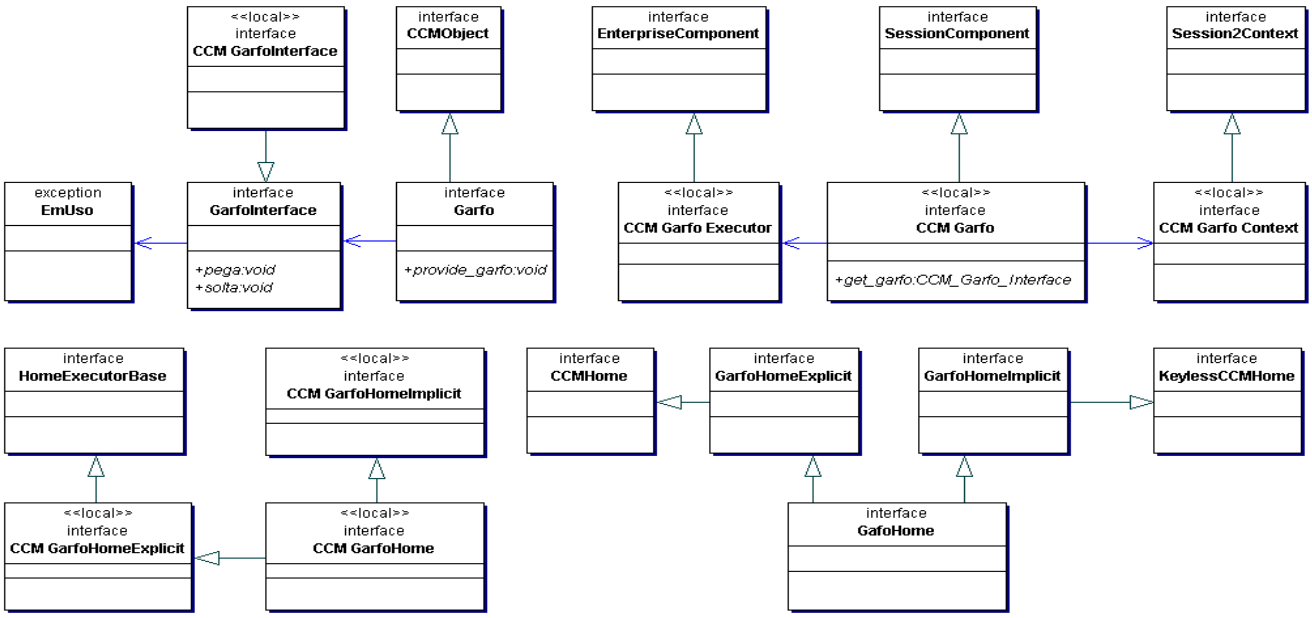

Figura 5-5 - Interfaces geradas em IDL equivalente para o componente Garfo

Os componentes Observador e Filosofo têm portas de comunicação assíncronas usadas para trafegar o evento "NovoEstado". Em IDL estendida, "NovoEstado" é declarado como um "*eventtype*", depois de convertido para IDL equivalente, a declaração do *eventtype* é convertida para *valuetype* e é gerada a interface *NovoEstadoConsumer* que é herdada pelas interfaces que atuam como *supplier* (porta emissora de eventos) e *consumer* (porta receptora de eventos). A hierarquia da superclasse *NovoEstadoConsumer* é representada na figura 5-6:

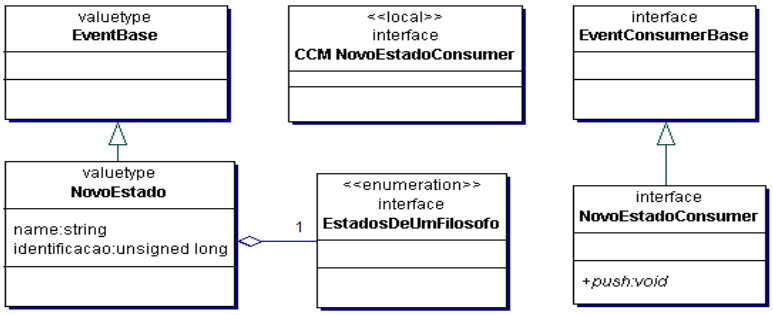

Figura 5-6 - Interfaces equivalentes para o evento NovoEstado.

As interfaces geradas na conversão do componente Observador são basicamente as mesmas geradas para o componente Garfo, a diferença está nas operações criadas nos executores (local e remoto) para atender à porta receptora de eventos. A figura 5-7 ilustra a versão estendida da interface:

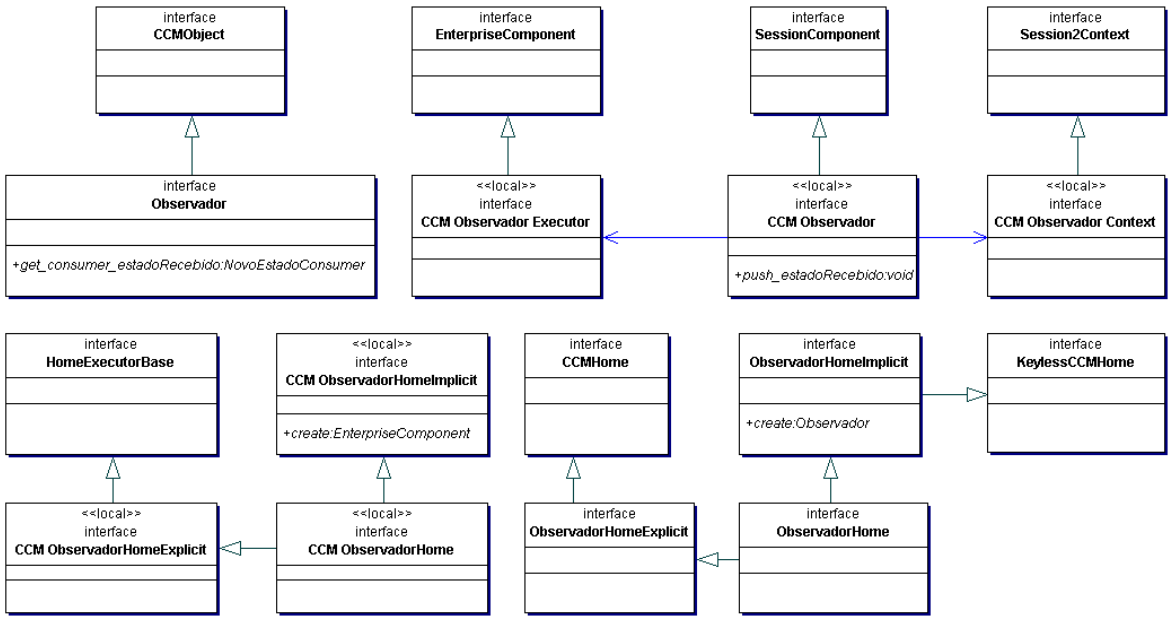

Figura 5-7 - Interfaces geradas em IDL equivalente para o componente Observador

A porta "*estadoRecebido*" também declara uma interface "*PushConsumer*" que especializa também a interface "*NovoEstadoConsumer*" (que foi representada na figura 5-6). O nome da interface também é "*NovoEstadoConsumer*" mas é declarada no módulo "*ObservadorEventConsumers*".

Para o componente Filosofo, o conjunto de interfaces geradas é muito semelhante ao conjunto gerado para o componente Observador.

A interface criada para a porta emissora de eventos, também chamada "*NovoEstadoConsumer*" e declarada no módulo "*FilosofoEventConsumers*", esta interface especializa "*PushConsumer*" e "*NovoEstadoConsumer*" (que foi representada na figura 5-6) .

Tanto as facetas quanto as portas emissoras de eventos têm operações especialmente declaradas na interface equivalente do componente e na interface contexto. As interfaces geradas para o componente Filosofo estão representadas na figura 5-8:

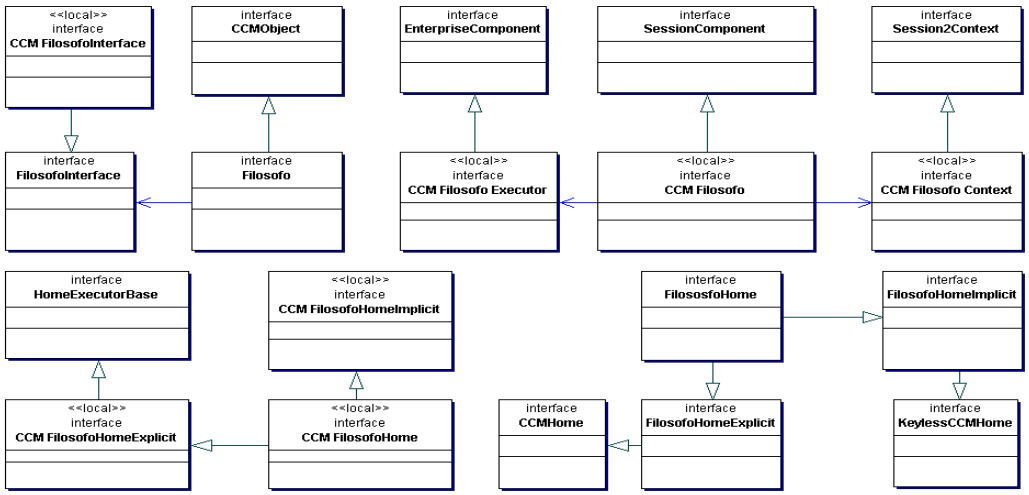

Figura 5-8 - Interfaces geradas em IDL equivalente para o componente Filosofo.

A geração de interfaces é somente parte do trabalho de criação do componente. Depois que as declarações em IDL equivalente foram geradas, é preciso gerar a implementação do componente propriamente dito, permitindo que as interfaces internas possam interagir com os elementos que compõem a arquitetura do contêiner e que o contêiner tenha "conhecimento" dos *servants* que implementam as interfaces de *callback*.

Os *servants* que implementam as interfaces internas devem ser gerados pelos utilitários do contêiner. Seu código é responsabilidade do implementador do contêiner e não deve ser alterado pelo desenvolvedor do componente, devido aos pormenores da arquitetura interna do contêiner (por exemplo: operações disponíveis em *singletons*, forma como os *servant managers* controlam a instanciação de cada objeto que participa da execução de um componente são decisões de projeto únicas para cada contêiner).

Sobre este aspecto a especificação também é intrigante. A interface *CCMObject* (que é herdada pela interface equivalente do componente) declara a operação *configuration_complete* que deve ser implementada pelo desenvolvedor do componente ferindo o "isolamento" do código gerado para implementar a interface.

Da forma como a especificação está escrita, o desenvolvedor do componente deve editar o código do *servant* que implementa as interfaces equivalentes e codificar a operação citada.

Para que isso não acontecesse, a operação *configuration_complete* deveria ser declarada também na interface *EnterpriseComponente* como uma operação de *callback* permitindo que possa ser acionada por um cliente e ainda assim evitando que a implementação da interface interna seja editada pelo desenvolvedor.

No protótipo, assim como aconteceu com a compilação de IDL, a geração de código não foi feita de forma automatizada, o código que deveria ser gerado por uma ferramenta foi escrito manualmente seguindo-se as orientações da especificação CCM e a arquitetura do protótipo.

5.5 AMBIENTE DE CODIFICAÇÃO E EXECUÇÃO DO PROTÓTIPO E DA APLICAÇÃO DE EXEMPLO.

O protótipo e a aplicação de exemplo foram codificados em um computador PC com processador Intel Pentium 3 com clock de 1.1 GHz e 256 MB de memória RAM usando o sistema operacional Microsoft Windows 2000. Ele foi construído sobre a versão 1.3.0 de um ORB de código aberto chamado OpenORB [OOR06] e compilado usando-se a versão 1.4.0 da linguagem Java (JDK).

A escolha pelo OpenORB aconteceu porque, como a linguagem de programação escolhida foi Java, o ORB deveria preferencialmente ser escrito também em Java para evitar surpresas relacionadas a configuração. O ORB deveria ser de código aberto e estar disponível para download gratuito para evitar problemas relacionados com direitos autorais e principalmente, deveria oferecer um serviço de nomes e um serviço notificações funcional. O OpenORB atendeu a todas estas características.

Antes de selecionar o OpenOrb, foi testado o ORB oferecido como parte do JDK 1.4.0 [SUN02b] que foi logo descartado por não oferecer um serviço de eventos ou notificações. Outro ORB verificado foi o JavaORB [DOG06] que no início da confecção do protótipo apresentava problemas para a configuração do serviço de notificações. O Borland Visibroker [BOR06] também foi cogitado com ouma opção, mas também foi descartado porquê, além de ser um produto comercial exige que o código seja realizado de forma não padrão (o ORB oferece operações para facilitar a obtenção de referência a POAs e a objetos remotos, mas isso torna o código da aplicação dependente do ORB).

A aplicação de exemplo foi executada com 2 até 10 filósofos, o comportamento interno de cada um dos componentes (filósofo, garfo e observador) foi acompanhado durante todo o seu ciclo de vida, para tanto foi usado a ferramenta de depuração oferecida pelo IDE eclipse para acompanhar passo a passo cada instância de objeto que representava cada um dos componentes, portas e elementos internos do contêiner. O objetivo desta atividade foi verificar a aderência entre o comportamento dos componentes e do contêiner com o que foi descrito anteriormente no capítulo 4 e sessão 5.3.

Assim que o comportamento do protótipo foi verificado (levando em conta que nem todas as operações descritas na especificação foram implementadas), a aplicação de exemplo foi executada também sem o apoio do IDE para verificar seu comportamento fora do ambiente controlado. Neste caso, a aplicação foi executada com sucesso com até 100 filósofos (como não é objetivo do trabalho verificar o desempenho do protótipo e sim sua aderência à especificação, não foi tentado identificar os limites do aplicativo).

5.6 CONCLUSÕES

A aplicação usada para teste foi adequada, os componentes gerados dispõem de todos os tipos de portas e foi possível testar: a obtenção de referências por meio do *HomeFinder*, o estabelecimento de conexões entre componentes e a troca de mensagens síncronas e assíncronas entre os componentes.

Cada componente manteve o seu estado durante toda a execução da aplicação de exemplo, sendo possível verificar que o conjunto de objetos CORBA que formavam os componentes foi preservado para cada um dos componentes instanciados.

O protótipo foi confeccionado para que o código que o desenvolvedor efetivamente tem que implementar (os executores locais) ficasse totalmente isolado do código gerado (no caso do protótipo, o código que implementa os elementos internos do container não foi gerado)

A parte da implementação que teria sido gerada se o protótipo oferecesse um utilitário de compilação de IDL3 ficou isolada da implementação do desenvolvedor do componente de forma que somente os executores locais seriam codificados (se um gerador de código tivesse sido desenvolvido).

Não foi oferecida nenhuma facilidade além do objeto contexto para que o executor local tivesse acesso aos recursos do contêiner, permitindo verificar o total isolamento oferecido pelo modelo de componentes com relação a arquitetura CORBA.

6 CONCLUSÕES

Este trabalho abordou o modelo de componentes CORBA sobre o ponto de vista do ambiente de execução e da estrutura interna dos componentes CCM e não sobre o ponto de vista da criação de componentes, para tanto, um protótipo funcional de um contêiner para componentes de sessão do tipo estendido foi efetivamente implementado e testado com uma aplicação de exemplo.

O uso do protótipo e a interpretação da especificação permitiram observar o comportamento de uma aplicação contendo componentes CCM com todos os tipos de portas previstos na especificação.

Constatou-se que o modelo de componentes CORBA oferece mais flexibilidade que os modelos EJB e MTS (.NET) pelo fato (entre outros) de poder ser executado em várias plataformas como o modelo EJB e de suportar várias linguagens de programação como no caso do modelo MTS.

CCM oferece mais categorias de componentes (são cinco categorias de componentes estendidos fora as categorias de componentes básicos) e cada categoria oferecida permite a declaração de portas de comunicação assíncronas (diferente de EJB que tem um tipo de componente específico para a troca de mensagens).

O modelo CCM oferece mais flexibilidade também quanto ao comportamento dos componentes, ele permite customizar quase que inteiramente o comportamento dos componentes através dos descritores de instalação e das regras de composição declaradas em CIDL fazendo com que, eventualmente, componentes da mesma categoria se comportem de formas diferentes através do ajuste da estratégia adotada para o gerenciamento da memória, do uso dos executores locais ou da forma como são codificadas as operações de *callback*.

Contudo, o modelo descrito na especificação para a criação das interfaces que dizem respeito às portas emissoras e receptoras de eventos poderia ser ajustado para melhorar a semântica, oferecendo um melhor entendimento dos papéis desempenhados pelas interfaces geradas.

Na versão implementada no protótipo, as interfaces "*consumer*" de cada componente são declaradas como "especializações" da interface "*consumer*" gerada para o tipo de evento que as portas recebem ou enviam, esta opção fez com que os objetos CORBA envolvidos na troca de mensagens fossem todos "*proxies*" (papel que efetivamente assumem frente ao serviço de notificações).

6.1 PAPEL DO MODELO DE COMPONENTES CORBA NO MERCADO DE SOFTWARE.

O modelo de componentes CORBA dificilmente alcançará o mesmo nível de popularidade dos demais modelos junto ao mercado de software. Enquanto o modelo de componentes EJB é maduro e amplamente testado e o modelo de componentes .NET é bastante atrativo para a comunidade de desenvolvedores que confeccionam aplicativos para serem executados em especial no sistema operacional Microsoft Windows. Ambos são

modelos amadurecidos, testados amplamente e contam com produtos que facilitam o processo de desenvolvimento além de oferecem interfaces visuais amigáveis e intuitivas para a confecção, empacotamento e instalação de componentes.

Por outro lado, o modelo de componentes CORBA embora robusto, não conta até o momento com nenhuma ferramenta que ofereça maior produtividade para o desenvolvimento de componentes CORBA, isso se explica devido ao fato do modelo de componentes CORBA apresentar uma visão diferenciada para um tipo de arquitetura para qual os grandes fabricantes de software como IBM, Sun, BEA ou Microsoft já investiram muitos recursos oferecendo soluções sobre outros modelos.

Por tratar-se de um modelo robusto e formalmente definido, o modelo CCM pode assumir o papel de modelo de referência para a evolução dos demais modelos em especial quanto a utilização formal de portas (isolando do desenvolvedor a necessidade de se preocupar com a criação de tipos especiais de componentes quando necessita modelar uma operação que se comporta de forma assíncrona) e da maior abstração que oferece ao analista de sistema no momento em que identifica o seu relacionamento existente entre os componentes da aplicação (ou entre os demais componentes disponíveis na corporação ou fora dela).

6.2 ARQUITETURAS ORIENTADAS A SERVIÇOS E O MODELO CCM

Assim como os modelos de componentes foram uma evolução natural para tecnologias para objetos distribuídos, a evolução natural para os modelos de componentes é na direção das arquiteturas orientadas a serviços (SOA – *Service Oriented Architecture*) [ERL04], em especial para serviços *Web* (*Web Services*) [DEI03][VIN04].

Neste contexto, o modelo de componentes CCM deve ocupar a mesma posição que os demais modelos de componentes, o de *back-end* para serviços *Web* ou demais arquiteturas orientadas a serviço [VIN04].

As interfaces oferecidas pelos serviços *Web* são representadas na forma de WSDL[CHR01][DEI03] assim como ás interfaces de componentes CORBA que são representadas na forma de IDL.

Para que componentes CORBA possam expor suas interfaces como serviços *Web*, pode-se converter as declarações das interfaces escritas em IDL para representação em WSDL. A forma como a conversão deve ser realizada foi especificada pela OMG no início de 2005 [VIN04][OMG05].

O envio de requisições às operações expostas pelos serviços *Web* é feita sobre o protocolo SOAP [DEI03], a especificação [OMG05] também descreve como deve ser feita a representação dos tipos de dados CORBA em SOAP.

Usando-se a especificação [OMG05], pode-se desenvolver *proxies* [GAM00] que permitam aos componentes CORBA exporem suas interfaces como serviços *Web*.

Em um futuro trabalho pretende-se descrever, implementar e testar uma "ponte" entre o modelo de componentes CORBA e os serviços *Web* baseado no modelo de interação CORBA [OMG02b], de forma semelhante às pontes e visões de componentes oferecidas entre componentes EJB e componentes CCM, como descrito no capítulo 3.4 deste documento.

REFERÊNCIAS BIBLIOGRÁFICAS

[BOL02] BOLTON, Fintan. *Pure Corba* – A code-Intensive Premium Reference. United States of America, Sams Publishing, 2002.

[BOX98] BOX, Don. *Q&A ActiveX/COM*. Microsoft Systems Journal. mar. 1998.

[BOR06] BORLAND. *Borland Visibroker*. (consultado em 2006). http://www.borland.com/us/products/visibroker/index.html

[CHR01] CHRISTENSEN, E. et al, (consultado em 2006) *Web Services Description Language (WSDL) 1.1*, W3C, Mar. 2001; www.w3.org/TR/wsdl.html.

[CIC99] CICALESE, Cyntia D. T., ROTENSTREICH, Shmuel. Behavioral Specification of Distributed Software Component Interfaces. *IEEE Computer* . s.l. [United States]: 46-53. jul. 1999.

[DEI03] DEITEL, Harvey et al, *Web services*: a technical introduction, New Jersey, Prentice Hall, 2003.

[ERL04] ERL, Thomas , *Service-oriented architecture*: a field guide to integrating XML and Web services, Nova Jersey, Prentice Hall Professional , 2004.

[FAR98] FARLEY, Jim, *JAVA Distributed Computing*. United States of America, O´Reille & Associates, 1998.

[GAM00] GAMA, Erich et all. *Padrões de Projeto* – Soluções Reutilizáveis de Software Orientado a Objetos.Porto Alegre, Bookman, 2000.

[HAR01] BOHME, Harald, NEUBAUER, Bertram, STOINSKI, Frank.*Project:P924 CCM container implementation* Deliverable 4. out 2001.

[HOR01] HORSTMANN, Cay S., CORNELL, Gary. *Core Java* - Volume II – Recursos Avançados. Trad. João Eduardo Nóbrega Tortello. São Paulo, Makron Books, 2001. p.223-78

[ICM06] ICMG. *K2-CCM Container*. (consultado em 2006) http://www.icmgworld.com/corp/ccm/ccm.overview.asp

[DOG06] DOG Distributed Object Group, (consultado em 2006). *JavaORB version 2.2.7*, http://dog.team.free.fr/details_javaorb.html

[KIR97] KIRTLAND, Mary, *Object-Oriented Software Development Made Simple with COM+ Runtime Services.* Microsoft Systems Journal . Nov. 1997.

[KIR97a] KIRTLAND, Mary, (2002) The COM+ Programming Model Makes it Easy to Write Components in Any Language. Microsoft Systems Journal Dez. 1997.

[MAR00] MARVIE, Raphael, MERLE, Philippe, GEIB, Jean-Marc, *Towards a Dynamic CORBA Component Platform.*set.2000 International Symposium on Distributed Objects and Applications p.305

[MEY99] MEYER, Bertrand. *On To Component*. IEEE Computer, jan. 1999, p. 130-40

[LIF02] LIFL, *OPENCCM The Open CORBA ComponentModel Platform*. www.objectweb.org/OpenCCM

[OMG99] OMG, *The Commom Object Request Broker: Architecture and Specification*, rev. 2.3.1, 1999 p.11.1-11.62

[OMG99a] OMG, Object management Group. *CORBA Components*. Vol I.. ago. 1999. http://www.omg.org/cgi-bin/doc?orbos/99-07-01

[OMG02] OMG, Object management Group. *CORBA Components*. jun. 2002. ver. 3,0. http://www.omg.org/cgi-bin/doc?formal/02-06-65

[OMG02a] OMG, Object management Group. *Notification Service Specification*. jun. 2002. ver. 1,1. http://www.omg.org/cgi-bin/doc?formal/04-10-11

[OMG02b] OMG, Object management Group. *Commom Object Request Broker Architecture: Core Specification*. dez. 2002. ver. 3.0. http://www.omg.org/cgi-bin/doc?formal/02-12-06

[OMG02c] OMG, Object management Group. *CCM Implementations*. jan. 2002. http://www.omg.org/cgi-bin/doc?ptc/02-02-03

[OMG02d] OMG, Object management Group. *Persistent State Service Specification*. set. 2002. ver. 2.0 .http://www.omg.org/cgi-bin/doc?formal/02-09-06

[OMG04] OMG, Object management Group. *Event Service Specification*. out. 2004. ver. 1.2. http://www.omg.org/cgi-bin/doc?formal/04-10-02

[OMG05] OMG, Object management Group. *CORBA to WSDL/SOAP Interworking Specification*. fev. 2005. ver. 1.0. http://www.omg.org/cgi-bin/doc?formal/05-02-01

[OOR06] OPEN ORB. *The Community OpenORB Project*. (acessado em 2006). http://openorb.sourceforge.net/

[ORF98] ORFALI, Robert, HARKEY, Dan. *Client/ Server Programing wit h JAVA and CORBA*. 2. ed. United States of America, John Wiley & Sons, 1998.

[PIL99] PILHOFER, Frank, (consultado em 2006) *Design and Implementation of the Portable Object Adapter.* Jun. 1999. http://fpx.de/fp/Uni/Diplom/diplom.html

[PIL02] PILHOFER, Frank, (consultado em 2006) *Writing andUsing CORBA Components.* Jan. 2002. www.fpx.de/MicoCCM/download/mico-ccm.pdf

[ROF99] ROFAIL, Ash, *Understanding MTS Components.*Visual Basic Programmers Journal. Jun. 1999.

[ROM02] ROMAN, Ed . M*astering Enterprise Java Beans*, 2. ed. United States of America, John Wiley & Sons. 2002.

[SER99] SERAIN, Daniel. *Middleware*. Trad. Iain Craig. s.l.[United Kingdon], Springer, 1999.

[SIE00] SIEGEL, Jon, *CORBA 3 Fundamentals and Programing*. 2. ed.United States of America, John Wiley & Sons, 2000.

[SUN02] SUN. *The Java 2 Enterprise Edition Developer´S Guide. Ver. 1.2.1.* www.sun.com

[SUNb] SUN. *Enterprise Java Beans Specification. Ver.2.1* www.sun.com

[SUN02b] SUN. *CORBA Technology and the Java 2 Platform, Standard Edition*.(acessado em 2006). http://java.sun.com/j2se/1.4.2/docs/guide/corba/index.html

[SZY02] SZYPERSKI, Clements, *Component Software* – Beyond Object-Oriented Programming. Second edition. Addison-Wesley, Harlow. Inglaterra. 2002.

[SZY03] SZYPERSKI, Clements, *Component Tecnology – What, Where, and How?* Proceedings of the 25th International Conferenceon Software Engineering (ICSE'03)

[VAU01] VAWTER, Chad, ROMAN, Ed, *J2EE vs. Microsoft.NET* – A Comparison of building XML-based web-services. Jun 2001.

[VIN98] VINOSKI, Steve, *New Features for CORBA 3.0*, Communications of the ACM, Vol.41, No.10, Out.1998.

[VIN04] VINOSKI, Steve, *CORBA in a loosely coupled world*, Set. 2004; www.looselycoupled.com/opinion/2004/vinos-corba-infr0908.html.

ANEXO 1 - DECLARAÇÃO DOS COMPONENTES EM IDL ESTENDIDA

GARFO.IDL3

```
#include "../../ccm/Components.idl"
#pragma prefix "exemplos.ccm.cfsm.com.br"

module filosofos {
      module garfo{
            exception EmUso{};
            interface GarfoInterface {
                  void pega() raises (EmUso);
                  void solta();
            };
            component Garfo {
                  provides GarfoInterface garfo;
            };
            home GarfoHome manages Garfo {};
      }; // module garfo
};  // module filosofos
```

OBSERVADOR.IDL3

```
#include "garfo.idl3"
#pragma prefix "exemplos.ccm.cfsm.com.br"
module filosofos {
      module observador{
            enum EstadosDeUmFilosofo {    COMENDO, PENSANDO,
                                          COM_FOME, DORMINDO };
            eventtype NovoEstado {
                  public string name;
                  public unsigned long identificacao;
                  public EstadosDeUmFilosofo estado;
            };

            component Observador {
                  consumes NovoEstado estadoRecebido;
            };
            home ObservadorHome manages Observador{};
      };
};
```

FILOSOFO.IDL3

```
#include "observador.idl3"
#pragma prefix "exemplos.ccm.cfsm.com.br"
module filosofos {
            component Filosofo {
                  attribute string nome;
                  attribute unsigned long tempoPensando;
                  attribute unsigned long tempoComendo;
                  attribute unsigned long identificacao;

                  void comendo();
                  void pensando();
                  void com_fome();
                  void dormindo();

                  uses GarfoInterface maoEsquerda;
                  uses GarfoInterface maoDireita;

                  publishes NovoEstado estadoEmitido;
            };

            home FilosofoHome manages Filosofo {
                  factory create(   in string name,
                                    in unsigned long identificacao);
            };

};
```

ANEXO 2 – DECLARAÇÃO DOS COMPONENTES EM IDL EQUIVALENTE

GARFO.IDL

```
#include "../../ccm/Components.idl"
#pragma prefix "exemplos.ccm.cfsm.com.br"

module filosofos {
            exception EmUso{};

            interface GarfoInterface {
                  void pega() raises (EmUso);
                  void solta();
            };

            local interface CCM_GarfoInterface : GarfoInterface { };

            interface Garfo : Components::CCMObject {
                  GarfoInterface provide_garfo();
            };

            local interface CCM_Garfo_Executor:
                              Components::EnterpriseComponent { };

            local interface CCM_Garfo: Components::SessionComponent{
                  CCM_GarfoInterface get_garfo();
            };

            interface GarfoHomeExplicit : Components::CCMHome {};
            interface GarfoHomeImplicit : Components::KeylessCCMHome {
                  Garfo create() raises(Components::CreateFailure);
            };
            interface GarfoHome : GarfoHomeExplicit, GarfoHomeImplicit {
};

            local interface CCM_GarfoHomeExplicit :
                              Components::HomeExecutorBase {};
```

```
        local interface CCM_GarfoHomeImplicit{
              Components::EnterpriseComponent create()
                    raises (Components::CCMException);
        };

        local interface CCM_GarfoHome :
              CCM_GarfoHomeImplicit, CCM_GarfoHomeExplicit{};
};
```

OBSERVADOR.IDL

```
#include "../../corba/CosNotifyComm.idl"
#include "garfo.idl"
#pragma prefix "exemplos.ccm.cfsm.com.br"

module filosofos {

      enum EstadosDeUmFilosofo { COMENDO, PENSANDO, COM_FOME, DORMINDO };

      valuetype NovoEstado:Components::EventBase {
            public string name;
            public unsigned long identificacao;
            public EstadosDeUmFilosofo estado;
      };

      interface NovoEstadoConsumer:Components::EventConsumerBase{
            void push (in NovoEstado the_NovoEstado);
      };

      local interface CCM_NovoEstadoConsumer {
            void push (in NovoEstado ev);
      };

      module ObservadorEventConsumers {
            interface NovoEstadoConsumer;
      };

      interface Observador : Components::CCMObject {
            NovoEstadoConsumer get_consumer_estadoRecebido();
      };

      module ObservadorEventConsumers {
            interface NovoEstadoConsumer:
                  CosEventComm::PushConsumer,
                  Components::EventConsumerBase,
                  filosofos::NovoEstadoConsumer{};
      };
```

```
        local interface CCM_Observador_Executor:
                        Components::EnterpriseComponent { };

        local interface CCM_Observador: Components::SessionComponent{
              void push_estadoRecebido (in NovoEstado ev);
        };

        local interface CCM_Observador_Context:
              Components::Session2Context{};

        interface ObservadorHomeExplicit : Components::CCMHome {};

        interface ObservadorHomeImplicit : Components::KeylessCCMHome {
              Observador create() raises(Components::CreateFailure);
        };

        interface ObservadorHome :
              ObservadorHomeExplicit, ObservadorHomeImplicit { };

        local interface CCM_ObservadorHomeExplicit :
              Components::HomeExecutorBase {};

        local interface CCM_ObservadorHomeImplicit{
              Components::EnterpriseComponent create() raises
              (Components::CCMException);
        };

        local interface CCM_ObservadorHome :
              CCM_ObservadorHomeImplicit, CCM_ObservadorHomeExplicit{};

};
```

FILOSOFO.IDL

```
#include "observador.idl"
#pragma prefix "exemplos.ccm.cfsm.com.br"

module filosofos {
      module FilosofoEventConsumers {
            interface NovoEstadoConsumer;
      };

      interface Filosofo : Components::CCMObject {
            attribute string nome;
            attribute unsigned long tempoPensando;
            attribute unsigned long tempoComendo;
            attribute unsigned long identificacao;

            void comendo();
            void pensando();
            void com_fome();
            void dormindo();

            void connect_maoEsquerda ( in GarfoInterface conxn ) raises (
                  Components::AlreadyConnected,
                  Components::InvalidConnection);

            GarfoInterface disconnect_maoEsquerda( ) raises (
                  Components::NoConnection );

            GarfoInterface get_connection_maoEsquerda( );

            void connect_maoDireita ( in GarfoInterface conxn ) raises (
                  Components::AlreadyConnected,
                  Components::InvalidConnection);

            GarfoInterface disconnect_maoDireita( ) raises (
                  Components::NoConnection );

            GarfoInterface get_connection_maoDireita( );
```

```
      Components::Cookie subscribe_estadoEmitido (
            in NovoEstadoConsumer consumer ) raises (
                  Components::ExceededConnectionLimit);

      NovoEstadoConsumer unsubscribe_estadoEmitido(
            in Components::Cookie ck) raises (
                  Components::InvalidConnection );
      };

local interface CCM_Filosofo_Executor:
      Components::EnterpriseComponent{};

local interface CCM_Filosofo: Components::SessionComponent{
      attribute string nome;
      attribute unsigned long tempoPensando;
      attribute unsigned long tempoComendo;
      attribute unsigned long identificacao;

      void comendo();
      void pensando();
      void com_fome();
      void dormindo();
};

local interface CCM_Filosofo_Context: Components::Session2Context{

      GarfoInterface get_connection_maoEsquerda();
      GarfoInterface get_connection_maoDireita();

      void set_connection_maoEsquerda(in GarfoInterface mao);
      void set_connection_maoDireita(in GarfoInterface mao);
      void push_estadoEmitido(in NovoEstado ev);
};

interface FilosofoHomeExplicit : Components::CCMHome {
```

```
            Filosofo create(  in string name,
                              in unsigned long identificacao);
      };

      interface FilosofoHomeImplicit : Components::KeylessCCMHome {
            Filosofo create() raises(Components::CreateFailure);
      };

      interface FilosofoHome:FilosofoHomeExplicit,
                              FilosofoHomeImplicit { };

      local interface CCM_FilosofoHomeExplicit :
            Components::HomeExecutorBase {
                  Components::EnterpriseComponent create(
                        in string name,
                        in unsigned long identificacao);
      };

      local interface CCM_FilosofoHomeImplicit{
            Components::EnterpriseComponent create() raises (
                  Components::CCMException);
      };

      local interface CCM_FilosofoHome :
            CCM_FilosofoHomeImplicit,
            CCM_FilosofoHomeExplicit{};

      module FilosofoEventConsumers {
            interface NovoEstadoConsumer:
                  CosEventComm::PushSupplier,
                  filosofos::NovoEstadoConsumer {
                  void setProxyConsumer(
                        in CosEventComm::ProxyPushConsumer proxyConsumer);
            };
      };
}; // module filosofos
```

ANEXO 3 – COMPONENTE FILÓSOFO - CONTEXT

CCM_Filosofo_ContextImpl.java

```
package br.com.cfsm.ccm.exemplos.filosofos;

import org.omg.CORBA.LocalObject;
import org.omg.CORBA.Object;
import org.omg.Components.BadComponentReference;
import org.omg.Components.CCMHome;
import org.omg.Components.HomeRegistration;
import org.omg.Components.IllegalState;
import org.omg.Components.PersistenceNotAvailable;
import org.omg.Components.PolicyMismatch;
import org.omg.Components.Transaction.UserTransaction;
import org.omg.CosPersistentState.CatalogBase;
import org.omg.SecurityLevel2.Credentials;

import br.com.cfsm.ccm.exemplos.filosofos.FilosofoEventConsumers.NovoEstadoConsumer;
import br.com.cfsm.ccm.server.sessionContainer.*;
import br.com.cfsm.ccm.server.singletons.*;

/**
 * Se houvesse uma ferramenta de geração de código, este programa
 * teria sido gerado
*/
public class CCM_Filosofo_ContextImpl
      extends LocalObject
      implements CCM_Filosofo_Context {

      // Variáveis de instância que representam os receptáculos
      // do componente
      private GarfoInterface maoEsquerda = null;
      private GarfoInterface maoDireita = null;

      // Referencias necessários ao Contexto para que execute as
      // operações Oferecidas em suas interfaces
      private Object componentReference = null;
      private NovoEstadoConsumer supplierEstadoEmitido = null;

      // Construtor default
      public CCM_Filosofo_ContextImpl() { }

      // implementado è partir do mapeamento descrito na especificação
      public GarfoInterface get_connection_maoEsquerda() {
            return maoEsquerda;
      }

      public GarfoInterface get_connection_maoDireita() {
            return maoDireita;
      }

      //
      // Sempre que uma conexão for estabelecida ou terminada,
      // o contexto deve ser atualizado.
      // Estas operações foram incluídas para permitir o sincronismo
      // entre a interface externa e o contexto
```

```
//
public void set_connection_maoDireita(GarfoInterface mao) {
      maoDireita = mao;
}

public void set_connection_maoEsquerda(GarfoInterface mao) {
      maoEsquerda = mao;
}

public void set_supplier_EstadoEmitido(
            NovoEstadoConsumer supplierEstadoEmitido) {
      this.supplierEstadoEmitido = supplierEstadoEmitido;
}

// Operação que emite o evento
public void push_estadoEmitido(NovoEstado novoEstado) {
      supplierEstadoEmitido.push(novoEstado);
}

// Operações herdadas de Session2Context
public Object create_ref(String repid) {
      return ObjectsPOAList.getWithRepId(repid).
             create_reference_with_id(ObjectIdManager.get(),
      repid);
}

public Object create_ref_from_oid(byte[] oid, String repid) {
      return ObjectsPOAList.getWithRepId(repid).
             create_reference_with_id(oid, repid);
}

public byte[] get_oid_from_ref(Object objref)
      throws IllegalState, BadComponentReference {

      byte[] oid = null;

      try{
            oid = ObjectsPOAList.getWithReference(objref).
                  reference_to_id(objref);
      } catch (org.omg.PortableServer.POAPackage.WrongAdapter wa){
            wa.printStackTrace();
            throw new BadComponentReference();
      } catch (Exception e){
            e.printStackTrace();
      }
      return oid;
}

public Object get_CCM_object() throws IllegalState {
      throw new IllegalState();
}

public HomeRegistration get_home_registration() {
      return null;
}

public void req_passivate() throws PolicyMismatch {
```

```
        }

        public CatalogBase get_persistence(String catalog_type_id)
              throws PersistenceNotAvailable {
              throw new PersistenceNotAvailable();
        }

        public CCMHome get_CCM_home() {
              CCMHome home = null;
              try{
                    home =  DeployedHomesSingleton.
                            getCCMHomeUsingHomeName("FilosofoHome");
              }catch(Exception e){}

              return home;
        }

        //
        // Operações fora do escopo de implementação do protótipo
        //
        public Credentials get_caller_principal() {
              return null;
        }

        public boolean get_rollback_only() throws IllegalState {
              return false;
        }

        public UserTransaction get_user_transaction() throws IllegalState {
              return null;
        }

        public boolean is_caller_in_role(String role) {
              return false;
        }

        public void set_rollback_only() throws IllegalState {
        }
}
```

ANEXO 4 – COMPONENTE FILÓSOFO – EXECUTOR DA INTERFACE EQUIVALENTE

FilosofoImpl.java

```
package br.com.cfsm.ccm.exemplos.filosofos;

import org.omg.CORBA.*;
import org.omg.Components.*;

// servico de notificações
import org.omg.CosNotification.*;
import org.omg.CosNotifyChannelAdmin.*;
import org.omg.CosNotifyComm.*;

// classes de apoio ao conteiner
import br.com.cfsm.ccm.server.sessionContainer.*;
import br.com.cfsm.ccm.server.singletons.*;
import br.com.cfsm.ccm.server.cachedServices.*;
import br.com.cfsm.ccm.exemplos.filosofos.FilosofoEventConsumers.*;

public class FilosofoImpl extends FilosofoPOA {

      private String componentRepId =

                  "IDL:exemplos.ccm.cfsm.com.br/filosofos/Filosofo:1.0";

      // executores locais
      private CCM_FilosofoImpl ccm_FilosofoImpl = null;
      private CCM_Filosofo_ContextImpl ccm_Filosofo_ContextImpl = null;


      // porta emissora de eventos
      private EventChannel canalDeEventos = null;
      private ProxyPushConsumer proxyConsumer = null;
      private ProxyPushSupplier proxySupplier = null;
      private NovoEstadoConsumer pushSupplier = null;
      private org.omg.CORBA.IntHolder proxySupplierId =
              new org.omg.CORBA.IntHolder();
      private org.omg.CORBA.IntHolder proxyConsumerId =
              new org.omg.CORBA.IntHolder();
      private org.omg.CORBA.IntHolder eventChannelId =
              new org.omg.CORBA.IntHolder();

      public FilosofoImpl(){
            canalDeEventos = getEventChannel();
            proxyConsumer = getProxyConsumer();
            proxySupplier = getProxySupplier();
      }

      public String nome() {
            verifyLocalExecutor();
            return ccm_FilosofoImpl.nome();
      }
```

```
public void nome(String value) {
      verifyLocalExecutor();
      ccm_FilosofoImpl.nome(value);
}

public int tempoPensando() {
      verifyLocalExecutor();
      return ccm_FilosofoImpl.tempoPensando();
}

public void tempoPensando(int value) {
      verifyLocalExecutor();
      ccm_FilosofoImpl.tempoPensando(value);
}

public int tempoComendo() {
      verifyLocalExecutor();
      return ccm_FilosofoImpl.tempoComendo();
}

public void tempoComendo(int value) {
      verifyLocalExecutor();
      ccm_FilosofoImpl.tempoComendo(value);
}

public int identificacao() {
      verifyLocalExecutor();
      return ccm_FilosofoImpl.identificacao();
}

public void identificacao(int value) {
      verifyLocalExecutor();
      ccm_FilosofoImpl.identificacao(value);
}

public void comendo() {
      verifyLocalExecutor();
      ccm_FilosofoImpl.comendo();
}

public void pensando() {
      verifyLocalExecutor();
      ccm_FilosofoImpl.pensando();
}

public void com_fome() {
      verifyLocalExecutor();
      ccm_FilosofoImpl.com_fome();
}

public void dormindo() {
      verifyLocalExecutor();
      ccm_FilosofoImpl.dormindo();
}
```

```
	public void connect_maoEsquerda(GarfoInterface conxn)
		throws AlreadyConnected, InvalidConnection {
		verifyLocalExecutor();
		ccm_Filosofo_ContextImpl.set_connection_maoEsquerda(conxn);
	}

	public GarfoInterface disconnect_maoEsquerda() throws NoConnection
{
		verifyLocalExecutor();
		GarfoInterface maoEsquerda =
			ccm_Filosofo_ContextImpl.get_connection_maoEsquerda();
		if(maoEsquerda == null) throw new NoConnection();
		GarfoInterface tmp = maoEsquerda;
		ccm_Filosofo_ContextImpl.set_connection_maoEsquerda(null);
		return tmp;
	}

	public GarfoInterface get_connection_maoEsquerda() {
		verifyLocalExecutor();
		return ccm_Filosofo_ContextImpl.get_connection_maoEsquerda();
	}

	public void connect_maoDireita(GarfoInterface conxn)
		throws AlreadyConnected, InvalidConnection {
		verifyLocalExecutor();
		ccm_Filosofo_ContextImpl.set_connection_maoDireita(conxn);
	}

	public GarfoInterface disconnect_maoDireita() throws NoConnection {
		verifyLocalExecutor();
		GarfoInterface maoDireita =
			ccm_Filosofo_ContextImpl.get_connection_maoDireita();
		if(maoDireita == null) throw new NoConnection();
		GarfoInterface tmp = maoDireita;
		ccm_Filosofo_ContextImpl.set_connection_maoDireita(null);
		return tmp;
	}

	public GarfoInterface get_connection_maoDireita() {
		verifyLocalExecutor();
		return ccm_Filosofo_ContextImpl.get_connection_maoDireita();

	}

	public Cookie subscribe_estadoEmitido(
					NovoEstadoConsumer pushConsumer)
		throws ExceededConnectionLimit {

		try{
			if(pushSupplier == null){
				byte[] pushSupplierOid = ObjectIdManager.get();
				org.omg.PortableServer.POA poa =
					ObjectsPOAList.getWithObjectName(
						"NovoEstadoSupplier");
				org.omg.CORBA.Object o =
				poa.create_reference_with_id(pushSupplierOid,
```

```
"IDL:filosofos.exemplos.ccm.cfsm.com.br/filosofos/FilosofoEventConsumers/
NovoEstadoConsumer:1.0");
                        pushSupplier = NovoEstadoConsumerHelper.narrow(o);

      ComponentsInstancesList.setObject(this._object_id(),
                                              pushSupplierOid,
                                              pushSupplier);

            ccm_Filosofo_ContextImpl.set_supplier_EstadoEmitido(
                              pushSupplier);
                  }
            } catch (Exception e){
                  e.printStackTrace();
            }

            try{
                  proxySupplier.connect_any_push_consumer(
                        (PushConsumer)pushConsumer);
                  proxyConsumer.connect_any_push_supplier( pushSupplier
);
                  pushSupplier.setProxyConsumer(proxyConsumer);

            } catch (org.omg.CosEventChannelAdmin.AlreadyConnected e) {
                  e.printStackTrace();
            } catch (org.omg.CosEventChannelAdmin.TypeError e) {
                  e.printStackTrace();
            }

            org.omg.Components.CookieImpl the_Cookie =
                  new org.omg.Components.CookieImpl();

            the_Cookie.setCookieValue(null);

            return the_Cookie;
      }

      public NovoEstadoConsumer unsubscribe_estadoEmitido(Cookie ck)
            throws InvalidConnection {
            return null;
      }

      public IRObject get_component_def() {
            ComponentDef componentDef = null;
            try{
                  componentDef =
                        ComponentDefListSingleton.get(componentRepId);
            } catch(Exception e){
                  e.printStackTrace();
            }
            return componentDef;
      }

      public CCMHome get_ccm_home() {
            CCMHome theHome = null;
            try{
                  theHome =
                  DeployedHomesSingleton.getCCMHomeUsingComponentRepId(
```

```
                componentRepId);
        } catch(Exception e){
              e.printStackTrace();
        }
        return theHome;
}


public void remove() throws RemoveFailure {
}

public ComponentPortDescription get_all_ports() {
        return null;
}

// não há facetas neste componente
public Object provide_facet(String name) throws InvalidName {
        throw new InvalidName();
}

// não há facetas neste componente
public FacetDescription[] get_all_facets() {
        return new FacetDescription[0];
}

// não há facetas neste componente
public FacetDescription[] get_named_facets(String[] names)
        throws InvalidName {
        throw new InvalidName ();
}

public boolean same_component(Object object_ref) {
        return this.equals(
                ComponentsInstancesList.getComponentRef(
                    object_ref));
}

public Cookie connect(String name, Object connection)
        throws
              InvalidName,
              InvalidConnection,
              AlreadyConnected,
              ExceededConnectionLimit {
        return null;
}

public Object disconnect(String name, Cookie ck)
        throws InvalidName,
               InvalidConnection,
               CookieRequired,
               NoConnection {
        return null;
}
```

```
public ConnectionDescription[] get_connections(String name)
      throws InvalidName {
      return null;
}

public ReceptacleDescription[] get_all_receptacles() {
      return null;
}

public ReceptacleDescription[] get_named_receptacles(
                                                  String[] names)
      throws InvalidName {
      return null;
}

public EventConsumerBase get_consumer(String sink_name)
      throws InvalidName {
      return null;
}

public Cookie subscribe(
      String publisher_name,
      EventConsumerBase subscriber)
      throws InvalidName, InvalidConnection,
             ExceededConnectionLimit {
      return null;
}

public EventConsumerBase unsubscribe(String publisher_name,
                                     Cookie ck)
      throws InvalidName, InvalidConnection {
      return null;
}

public void connect_consumer(
      String emitter_name,
      EventConsumerBase consumer)
      throws InvalidName, AlreadyConnected, InvalidConnection {
}

public EventConsumerBase disconnect_consumer(String source_name)
      throws InvalidName, NoConnection {
      return null;
}

public ConsumerDescription[] get_all_consumers() {
      return null;
}

public ConsumerDescription[] get_named_consumers(String[] names)
      throws InvalidName {
      return null;
}

public EmitterDescription[] get_all_emitters() {
      return null;
}
```

```
	public EmitterDescription[] get_named_emitters(String[] names)
		throws InvalidName {
		return null;
	}

	public PublisherDescription[] get_all_publishers() {
		return null;
	}

	public PublisherDescription[] get_named_publishers(String[] names)
		throws InvalidName {
		return null;
	}

	public void configuration_complete() throws InvalidConfiguration {
	}

	public PrimaryKeyBase get_primary_key() throws NoKeyAvailable {
		throw new NoKeyAvailable();
	}

	private EventChannel getEventChannel(){
		EventChannel _canalDeEventos = null;
		try {

			org.omg.CORBA.Object o = TheORB.

                                        get().resolve_initial_references
                                        (
                                           "NotificationService");
			EventChannelFactory eventChannelFactory =
                                      EventChannelFactoryHelper.narrow(o);

			try {
                                _canalDeEventos =
eventChannelFactory.create_channel(
                                new Property[0],
                                new Property[0],
                                proxyConsumerId );

			} catch ( org.omg.CosNotification.UnsupportedQoS ex ) {
                                ex.printStackTrace( System.err );
			} catch ( org.omg.CosNotification.UnsupportedAdmin ex )
{
                                ex.printStackTrace( System.err );
			}

		} catch (Exception e) {
			e.printStackTrace();
		}
		return _canalDeEventos;
	}

	private ProxyPushConsumer getProxyConsumer(){
		ProxyPushConsumer _proxyConsumer = null;
		SupplierAdmin supplierAdmin =
```

```
                        canalDeEventos.default_supplier_admin();

            try {
                  _proxyConsumer  = ProxyPushConsumerHelper.narrow(

      supplierAdmin.obtain_notification_push_consumer(
ClientType.ANY_EVENT, proxyConsumerId ) );
            } catch ( org.omg.CosNotifyChannelAdmin.AdminLimitExceeded ex
) {
                  ex.printStackTrace();
            } catch ( org.omg.CORBA.BAD_PARAM ex ) {
                  ex.printStackTrace( System.err );
            }

            return _proxyConsumer ;
      }

      private ProxyPushSupplier getProxySupplier(){
            ProxyPushSupplier _proxySupplier = null;
            ConsumerAdmin consumerAdmin =
canalDeEventos.default_consumer_admin();

            try {
                  _proxySupplier  = ProxyPushSupplierHelper.narrow(
                  consumerAdmin.obtain_notification_push_supplier(
ClientType.ANY_EVENT, proxySupplierId));
            } catch ( org.omg.CosNotifyChannelAdmin.AdminLimitExceeded ex
) {
                  ex.printStackTrace();
            } catch ( org.omg.CORBA.BAD_PARAM ex ) {
                  ex.printStackTrace( System.err );
            }

            return _proxySupplier ;
      }

      private void verifyLocalExecutor(){
            byte[] oid =  this._object_id();
            if(ccm_FilosofoImpl == null){
                  ccm_FilosofoImpl =
                  (CCM_FilosofoImpl)ComponentsInstancesList.getExecutorLo
            cal(
                        this._object_id());
                  ccm_Filosofo_ContextImpl =
                  (CCM_Filosofo_ContextImpl)ComponentsInstancesList.getCo
                  ntext(this._object_id());
            }
      }

}
```