# Um Modelo para Deployment de Componentes em CORBA

*Maria Claudia Borges Barros*

**Dissertação de Mestrado**

Instituto de Computação
Universidade Estadual de Campinas

# Um Modelo para Deployment de Componentes em CORBA

**Maria Claudia Borges Barros**

Julho de 2003.

**Banca Examinadora**

- Prof. Dr. Edmundo Roberto Mauro Madeira (Orientador)
  IC – Instituto de Computação - UNICAMP
- Prof. Dr. Eleri Cardozo
  FEEC – Faculdade de Engenharia Elétrica e de Computação - UNICAMP
- Profa. Dra. Islene Calciolari Garcia
  IC – Instituto de Computação – UNICAMP
- Prof. Dr. Luiz Eduardo Buzato (Suplente)
  IC – Instituto de Computação – UNICAMP

**FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP**

Barros, Maria Claudia Borges

B278m Um modelo para deployment de componentes em CORBA / Maria Claudia Borges Barros – Campinas, [S.P. :s.n.], 2003.

Orientador: Edmundo Roberto Mauro Madeira

Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. CORBA (Arquitetura de computador). 2. Programação orientada a objetos (Computação). 3. Software – Desenvolvimento. 4. Software – Reutilização. I. Madeira, Edmundo Roberto Mauro. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título

# Um Modelo para Deployment de Componentes em CORBA

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Maria Claudia Borges Barros e aprovada pela Banca Examinadora.

Campinas, 13 de agosto de 2003.

Prof. Dr. Edmundo Roberto Mauro Madeira
IC – UNICAMP (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

# Resumo

Componentes são entidades de software reutilizáveis que podem ser combinadas para criar novas aplicações. Pode-se pensar no componente como um pedaço de software que implementa uma funcionalidade específica de forma completa, e provê interfaces capazes de alterar a execução e/ou a configuração dessa funcionalidade.

Nessa dissertação apresentamos um modelo para efetuar o deployment, ou seja, a instalação e ativação dos componentes, em CORBA. Avaliamos também as vantagens e desvantagens de possíveis implementações desse modelo.

# Abstract

Components are reusable software entities that can be combined in order to create new applications. Let's think a component as a piece of software that implements a specific functionality completely, and provides interfaces to change the execution and/or configuration of this functionality.

In this dissertation, a model to handle the deployment (instalation and activation) of CORBA Componentes is presented. Also, possible implementations for the model are evaluated.

# Sumário

# Lista de Tabelas

# Lista de Figuras

# Lista de Acrônimos

| Acrônimo | Significado |
|---|---|
| AAR | Assembly Archive |
| ANSA | Advanced Networked Systems Architecture |
| BOA | Basic Object Adapter |
| CAD | Component Assembly Descriptor |
| CAI | Component Assembly Instances |
| CAR | Component Archive |
| CCD | CORBA Component Descriptor |
| CCM | CORBA Component Model |
| CIDL | Component Implementation Definition Language |
| COM | Component Object Model |
| CORBA | Common Object Request Broker Architecture |
| CPF | Component Property File |
| DCOM | Distributed Component Object Model |
| DLL | Dynamic Linking Library |
| EJB | Enterprise Java Beans |
| FTP | File Transfer Protocol |
| IDL | Interface Definition Language |
| IIOP | Internet Inter-ORB Protocol |
| IOR | Interoperable Object Reference |
| MOF | Meta Object Facility |
| OMA | Object Management Architecture |
| OMG | Object Management Group |
| ORB | Object Request Broker |
| PECT | Prediction-Enabled Component Technology |
| POA | Portable Object Adapter |
| RFI | Request For Information |
| RFP | Request For Proposal |

| | |
|---|---|
| TAO | The ACE (Adaptative Communication Environment) ORB |
| XMI | XML Meta Data Interchange |
| XML | Extensible Markup Language |

# Capítulo 1
# Introdução

Durante o desenvolvimento de aplicações em grande escala, deve-se projetar a implementação de lógicas de negócio e a integração em uma arquitetura distribuída capaz de prover eventos, espaço de nomes, segurança, tolerância a falhas, e assim por diante. No entanto, essa é uma tarefa bastante complexa que pode requerer um esforço considerável, embora repetitivo. Uma maneira de acelerar o desenvolvimento das aplicações é considerar a utilização de ferramentas que gerem código automaticamente, tanto para a implementação das lógicas de negócio quanto para a integração na plataforma distribuída. Neste sentido, a tecnologia que tem sido mais amplamente desenvolvida e utilizada é a de componentes de software, como por exemplo: COM (Component Object Model) [20], DCOM (Distributed COM) [20] e Enterprise JavaBeans [6].

Um componente de software tem interfaces bem definidas e propriedades que facilitam a configuração e a conexão com outros componentes de software. Ao utilizar a tecnologia de componentes, o desenvolvedor da aplicação apenas efetua a ligação entre os componentes existentes a fim de produzir a semântica de negócio desejada. A infra-estrutura para comunicação distribuída, eventos, transações, segurança, espaço de nomes e outros serviços está encapsulada dentro dos componentes. Deste modo, fica mais fácil construir e dar manutenção em uma aplicação.

O OMG (Object Management Group) gerou uma especificação para Componentes em CORBA chamada CCM (CORBA Component Model) [5]. CCM provê suporte para a definição, geração de código, empacotamento, instalação e ativação de componentes em CORBA. Um ponto interessante é que o CCM permite interação e integração com o EJB (Enterprise JavaBeans) 1.1.

Uma parte importante do uso de componentes é a instalação e ativação dos mesmos (deployment), que consiste em instalar, instanciar e conectar os componentes, deixando-os em condições de atender às requisições dos clientes. No entanto, a especificação CCM não contém detalhes de como implementar o deployment de componentes CORBA. Portanto, essa

dissertação de mestrado propõe um modelo para a instalação e ativação de componentes CORBA, detalhando os participantes do modelo, o POA (Portable Object Adapter) para componentes, as mudanças na IOR (Interoperable Object Reference) para componentes e o processo de deployment. É mostrado ainda um caso de uso do modelo a título de exemplo, além de alguns testes realizados com o objetivo de provar conceitos e avaliar performance.

A dissertação está organizada em seis capítulos. O capítulo 2 apresenta os conceitos teóricos utilizados na dissertação: CORBA e, em particular, POA. O capítulo 3 mostra o conceito de Componentes e explora os trabalhos relacionados e o estado da arte na questão de Deployment de Componentes. No capítulo 4 apresentamos nosso Modelo de Deployment para Componentes. No capítulo 5 são discutidos questões de implementação do modelo proposto e os resultados obtidos. E finalmente, no capítulo 6 estão nossos comentários, conclusão e sugestões de trabalhos futuros.

# Capítulo 2
# Conceitos Básicos

## *2.1 OMG (Object Management Group)*

O OMG é um consórcio de companhias, universidades e centros de pesquisa da área de computação formado em 1989. Seu propósito é desenvolver padrões que possibilitem a interoperabilidade e portabilidade de aplicações distribuídas, que culminou nas arquiteturas CORBA (Common Object Request Broker Architecture) [2] e OMA (Object Management Architecture). Entre seus membros, mais de 800, estão algumas das maiores empresas comerciais da área de computação distribuída (IBM, HP, Digital, Iona, entre outras), e instituições de pesquisa (como a UNICAMP).

Diferente de outros consórcios, o OMG não produz software, apenas especificações. Para cada tópico de interesse, o OMG lança uma requisição por informação (Request for Information - RFI) e uma requisição por propostas (Request for Proposals - RFP). A especificação será montada com as idéias e tecnologias enviadas pelos membros em resposta às requisições. Esse processo tem muito valor, pois leva as entidades participantes a um consenso, e todos se beneficiam com os padrões gerados.

Todas as especificações são disponibilizadas de graça, para qualquer entidade (membro do OMG ou não), que deseje implementá-las. Como não há custos, várias companhias comerciais, universidades e laboratórios de pesquisa implementam CORBA. Todos querem estar alinhados com as tendências colocadas pelas grandes companhias de computação e universidades, o que torna a disseminação dos padrões mais rápida.

## 2.2 CORBA (Common Object Request Broker Architecture)

A plataforma descrita pela CORBA é composta pelo núcleo do ORB (Object Request Broker) e por suas interfaces de chamadas, permitindo a interação transparente entre os objetos clientes e servidores em um ambiente distribuído.

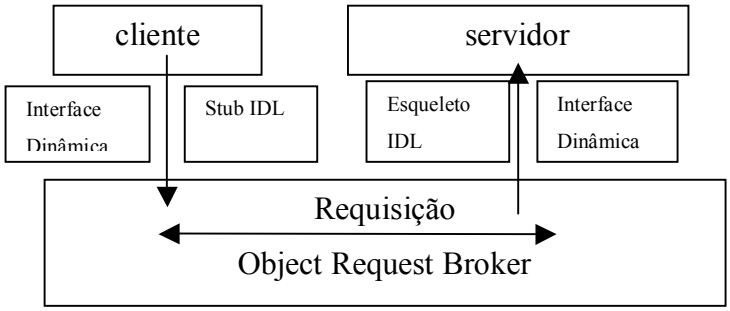


**Figura 1 – Fluxo da Requisição**

Utilizando o ORB, um objeto cliente pode, transparentemente, invocar um método em um objeto servidor (chamado de Servant), que pode estar na mesma máquina ou em qualquer ponto da rede (Figura 1).

O ORB recebe uma requisição com os parâmetros necessários e é responsável por achar o objeto que satisfaça essa requisição, invocar o método apropriado e retornar os resultados, caso necessário.

O OMG adotou como padrão uma linguagem orientada a objetos específica para definir as operações e as interfaces dos objetos. A IDL (Interface Definition Language) é uma linguagem puramente declarativa e é usada para definir interfaces e métodos de invocação para as linguagens suportadas pela CORBA. Atualmente C, C++[4], Smalltalk, Ada, Cobol, Phyton, Lisp e Java [3] já são suportadas e trabalhos de mapeamento estão sendo desenvolvidos para outras linguagens.

A estrutura descrita pela especificação CORBA é composta pelo núcleo do ORB e suas interfaces como mostra a Figura 2:

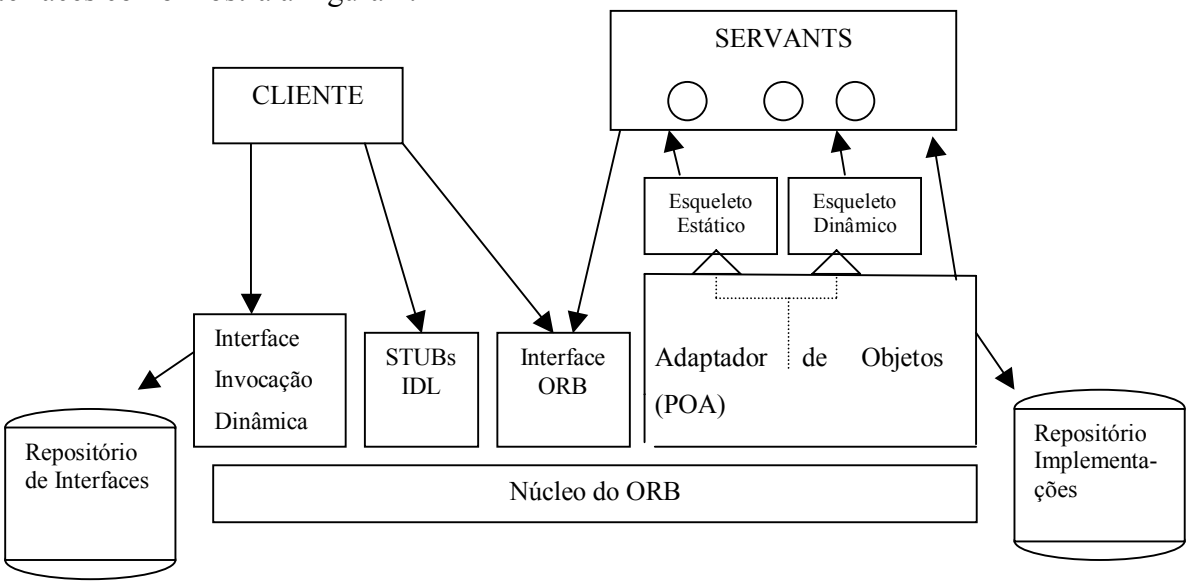


**Figura 2 – Arquitetura CORBA**

O **Núcleo do ORB** recebe as requisições dos clientes através da interface estática (**IDL-STUB**) ou da **Interface de Invocação Dinâmica** e localiza a implementação de objetos (servidor), transmitindo os parâmetros, transferindo o controle através de uma interface **Esqueleto-IDL estática** ou **dinâmica** e retornando as respostas, caso existam.

Os **STUBs** são compilados em IDL e contêm código para construir a chamada da operação com seus parâmetros em formatos de mensagens que poderão ser enviadas ao servidor. A **Interface de Invocação Dinâmica** permite descobrir o método (e/ou parâmetros) a ser chamado em tempo de execução. O cliente especifica o objeto a ser invocado e obtém a descrição do método no **Repositório de Interfaces** através de uma seqüência de chamadas.

Situado sobre o ORB, interligando serviços e aceitando requisições de serviços em benefício das implementações de objetos está o **Adaptador de Objetos**. Qualquer ORB proverá um adaptador de objetos padrão denominado BOA (Basic Object Adapter) e poderá suportar mais de um adaptador. Ele permite instanciar objetos, passar requisições para estes e atribuir referências de objetos. O adaptador também registra as classes e os objetos no **Repositório de Implementações**. O adaptador POA (Portable Object Adapter) apresenta um modelo mais elaborado, que consiste na criação de **Servants** para a execução dos objetos, e será descrito em detalhes na Seção 2.5.4.

O ORB possui dois repositórios:

- **Repositório de Implementações**: contém as informações que permitem localizar e ativar implementações de objetos.
- **Repositório de Interfaces**: provê o armazenamento consistente de definições de interfaces.

O **Esqueleto Estático** provê interfaces para cada serviço exportado pelo servidor. Assim como os STUBs, são criados através do compilador IDL.

A **Interface de Esqueleto Dinâmica** provê ligação (binding) para servidores que precisam manipular chamadas de métodos para objetos que não possuem Esqueletos Estáticos.

## *2.3 OMA (Object Management Architecture)*

A CORBA é excelente para conectar objetos, mas não aplicações. Para suprir a necessidade de um framework de integração em um ambiente distribuído, o OMG lançou a OMA.

A especificação OMA descreve um conjunto padrão de interfaces e funções que pode ser utilizado pelos objetos distribuídos. É dividida em quatro partes: ORB, Serviços de Objetos, Facilidades Comuns e Interfaces de Domínio, conforme mostrado na Figura 3.

Os Serviços CORBA provêem funcionalidades básicas, que quase todo objeto vai precisar como serviço de ciclo de vida, serviço de nomes, serviço de trader, entre outras. Já as Facilidades CORBA provêem serviços para as aplicações, facilitando a integração; como por exemplo facilidade de impressão, facilidades de gerenciamento de informação como MOF (Meta Object Facility) e XMI (XML Metadata Interchange), entre outros.

As Interfaces de Domínio provêem funcionalidades voltadas para aplicações de áreas específicas, como por exemplo Finanças, Manufatura, Telecomunicações, entre outras.

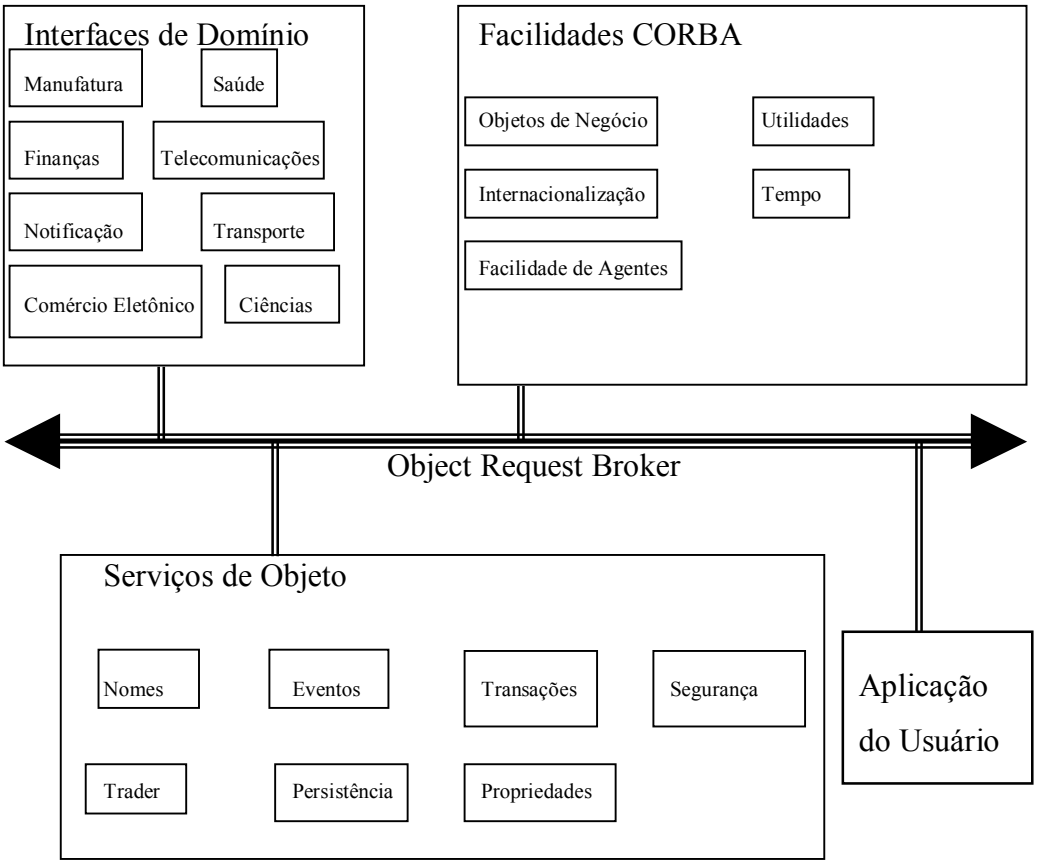


**Figura 3 – Arquitetura OMA**

## *2.4 Serviços CORBA*

A seguir, comentam-se alguns serviços CORBA relevantes para esse trabalho.

### 2.4.1. Serviço de Nomes

Permite que objetos localizem outros objetos pelo nome. Armazena a associação entre referências de objetos e nomes.

Pode formar uma federação com os serviços de nomes de outros domínios, compartilhando seus bancos de dados. Com isso, objetos pertencentes a um serviço de nomes remoto podem aparecer no resultado de uma consulta feita ao serviço de nomes local.

### 2.4.2. Serviço de Trader

É semelhante a um serviço de páginas amarelas, onde os objetos informam sobre suas funcionalidades. Através do trader, os objetos exportam "ofertas de serviços", identificadas por TipoDeServiço, TipoDeInterface e um conjunto de propriedades definidas por pares "nome-valor".

Ao fazer uma consulta ao trader, o cliente especifica o TipoDeServiço e/ou as propriedades que interessam, e recebe como resposta as referências de todos os objetos que atendem às especificações.

Os traders também podem formar federações, para efeitos de balanceamento de carga (load-balancing) ou para compartilhar as ofertas de serviço através de diferentes domínios.

### 2.4.3. Serviço de Eventos

Define um canal de eventos (event channel), que coleta e distribui eventos entre objetos que nada sabem um do outro.

O objeto que emite o evento é chamado de fornecedor (supplier), e o objeto que consome o evento é o consumidor (consumer).

É possível conectar um número ilimitado de fornecedores e consumidores a um canal de eventos, conforme ilustrado na Figura 4.

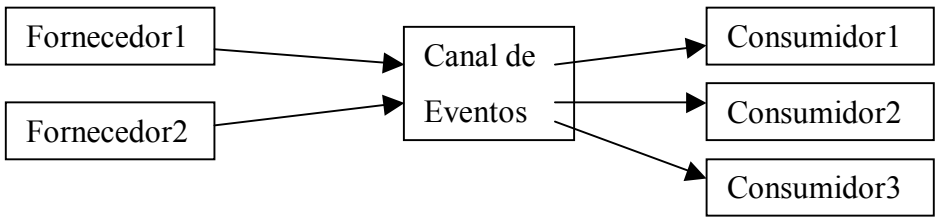


**Figura 4 – Canal de Eventos**

Existem quatro modelos de comunicação entre as entidades fornecedor, consumidor e canal de eventos:

- Modelo Push: os fornecedores enviam eventos para o canal de eventos, que por sua vez os repassa para os consumidores registrados;
- Modelo Pull: os consumidores solicitam eventos ao canal de eventos, que solicita os eventos ao fornecedor. O fornecedor daí, em resposta à solicitação, envia os eventos ao canal de eventos, que os repassa ao consumidor.

- Modelo Push/Pull: os fornecedores enviam eventos ao canal de eventos, que os armazena. Os eventos só serão repassados aos consumidores quanto eles os solicitarem;
- Modelo Pull/Push: o canal de eventos decide quando solicitar eventos aos fornecedores e quando repassá-los aos consumidores.

Os eventos carregam um dado do tipo Any, através do qual pode ser enviado qualquer tipo de informação (string, struct, entre outros).

## *2.5 Adaptadores de Objetos*

O Adaptador de Objetos (OA – Object Adapter) é a parte do ORB que interage com a implementação do objeto.

Antes de descrever os adaptadores de objeto padrões (BOA - *Basic Object Adapter* e POA – *Portable Object Adapter*), comentaremos o modelo computacional abstrato onde esses adaptadores atuam.

### 2.5.1. Modelo Abstrato do Adaptador de Objetos

A Figura 5 ilustra o Modelo Abstrato do Adaptador de Objetos.

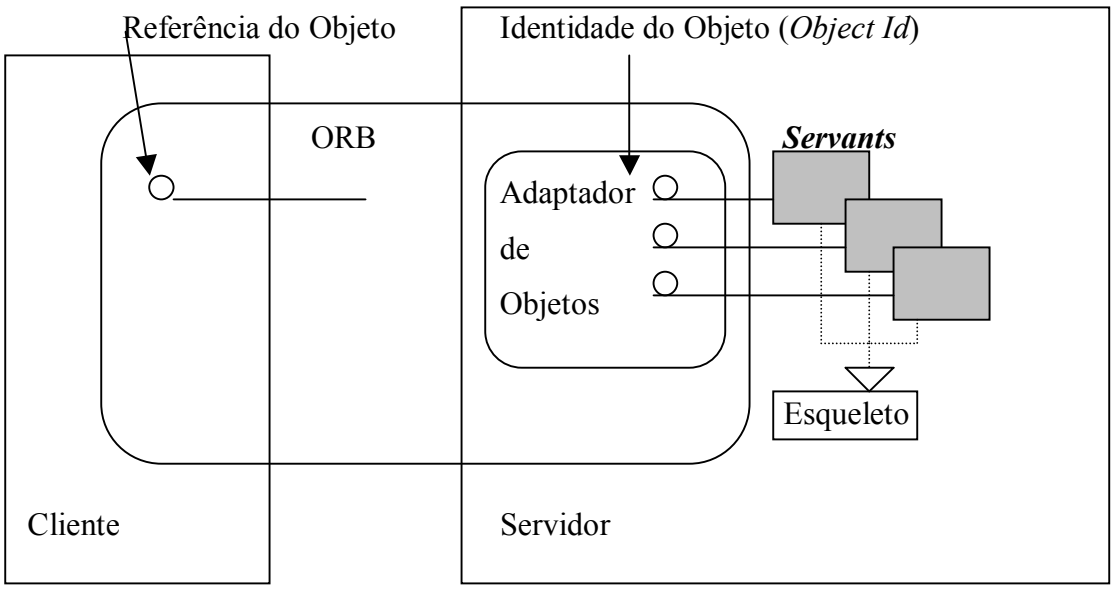


**Figura 5 – Modelo Abstrato do Adaptador de Objetos**

O **Cliente** obtém a referência de um objeto que implementa uma dada funcionalidade (através de um serviço de nomes ou do trader, por exemplo) e a seguir, envia requisições para o objeto (através do ORB), solicitando a execução de uma operação.

O **Objeto** é a entidade virtual que processa as requisições dos clientes e é formada por: uma identidade, uma interface e uma implementação. Do ponto de vista do cliente, é identificada pela referência do objeto. No contexto do servidor, o objeto é identificado pelo *objectId*, que é gerenciado pelo **Adaptador de Objetos**.

O **Servidor** é o contexto computacional onde a implementação de um objeto reside, normalmente corresponde a um processo. O ***Servant*** é uma instância de um objeto que processa uma requisição. O objeto é uma entidade virtual. Quando uma requisição chega, o ORB aloca/cria uma entidade real, o servant, para processá-la. O Servant é criado no contexto do processo servidor.

O **Esqueleto (*Skeleton*)** faz a interface entre o Servant e o Adaptador de Objetos. Normalmente é implementada na forma de uma classe base da qual o Servant deriva. Existem dois tipos: Esqueleto Estático e Esqueleto Dinâmico. O Adaptador de Objetos mantém o **Mapa de Objetos Ativos** que é uma tabela com as associações entre os objetos ativos e os respectivos Servants. A **Identificação do Objeto (*Object Id*)** é utilizada para identificar o objeto dentro do Adaptador de Objetos onde ele foi criado e registrado. Pode ser criada pelo próprio Adaptador de Objetos, ou pela implementação do objeto.

A **Ativação** faz com que um objeto seja capaz de processar as requisições vindas dos clientes. Implica na criação de um Servant e de uma associação do Servant ao objeto. A **Desativação** é o oposto da ativação, implica na destruição da associação entre o objeto e o Servant. O objeto continua existindo, mas o Servant pode ser destruído.

A **Encarnação (*Incarnation*)** é o ato de associar um servant a um objeto, de modo que o objeto passe a poder processar requisições. Já a **Eterealização (*Etherealization*)** é o oposto da Encarnação, e desfaz a associação entre o servant e o objeto.

Os termos **Ativação/Desativação** referem-se ao objeto. Significa tornar o objeto apto/inapto a receber requisições. Os termos **Encarnação/Eterealização** referem-se ao gerenciamento da associação entre o objeto e um Servant capaz de processar as requisições recebidas.

Durante a sua existência, o objeto pode encarnar em vários Servants diferentes. Por exemplo, o objeto é ativado para atender uma requisição, encarna em um Servant que processa a requisição, e ao final o objeto é desativado. Ao ser novamente ativado para atender mais uma requisição, o objeto encarna em outro Servant. Outro exemplo é o objeto encarnar em um Servant, a requisição ser processada, e o Servant ser eterealizado. Ao receber outra requisição, o objeto encarna em outro Servant que após o processamento da requisição é eterealizado.

Um Servant pode ser associado a vários objetos diferentes. Nesse caso, ele utilizará os *Object Ids* (contidos na referência) para distinguir entre um objeto e outro e efetuar o processamento da requisição de acordo. A Figura 6 [9] resume os cenários de Ativação/Desativação e Encarnação/Eterealização.


**Figura 6 – Cenários do Ciclo de Vida de um Objeto no POA**

### 2.5.2. Funcionalidades do Adaptador de Objetos

Um adaptador de objetos deve prover as seguintes funcionalidades:

- Demultiplexação das Requisições: distribui as requisições para os Servants correspondentes

- Despacho das Operações: transforma os parâmetros enviados na requisição em argumentos e invoca a operação correspondente do Servant através do Esqueleto.
- Ativação e desativação dos objetos: implica na encarnação e eterealização de Servants correspondentes
- Geração de referências para os objetos: gerar as referências dos objetos registrados nele. Essas referências devem ser utilizadas pelos métodos padrão do ORB para localizar os objetos.

### 2.5.3. BOA – Adaptador Básico de Objetos

A primeira versão da especificação CORBA definiu o **Adaptador Básico de Objetos** (BOA - *Basic Object Adapter*), como sendo o primeiro Adaptador de Objetos padrão. O BOA suporta quatro modelos de ativação dos Servidores:

- Servidor Não-compartilhado: um processo servidor para cada objeto.
- Servidor Compartilhado: um processo servidor suporta vários tipos de objetos
- Servidor Persistente: o processo servidor não é inicializado automaticamente pelo ORB. É iniciado por scripts que executam no momento do boot da máquina, e lêem seu estado persistente. Um Servidor Persistente pode ser Compartilhado ou Não-Compartilhado.
- Servidor Por-Operação: O objeto não é implementado por um único processo servidor, mas por um conjunto de processos servidores, um para cada operação do objeto. A maioria dos ORBs comerciais não suporta esse tipo de servidor.

A especificação do BOA era vaga e incompleta numa série de aspectos:

- Não definia um modo portável de associar Esqueletos com Implementações de Objetos: de qual classe a Implementação do Objeto deve derivar? A classe Esqueleto deveria servir de base para a Implementação do Objeto, de forma a possibilitar a comunicação entre o Servidor e o ORB?
- Falhava em descrever como as Implementações dos Objetos são registradas no BOA: pode ser implicitamente (o construtor da Implementação do Objeto se cadastra no BOA) ou explicitamente (a Implementação do Objeto chama um método do BOA).

- Falhava em descrever as funções necessárias para fazer um Servidor escutar as requisições: existem duas operações que habilitam o Servidor a processar as requisições (*impl_is_ready* e *obj_is_ready)*. Elas deveriam ser chamadas conforme o modelo de ativação desejado. Mas não está claro na especificação do BOA como isso deve ser feito, e cada vendedor implementou de um jeito.

Devido a esses problemas, cada vendedor implementou o seu "próprio" BOA, o que tornou os Servidores não–portáveis entre ORBs de fabricantes diferentes. Por isso, o OMG lançou a requisição por propostas (*Request For Proposals*) para o **Adaptador de Objetos Portável** [7] (POA – *Portable Object Adapter*) que substituiu o BOA em 1997.

### 2.5.4. Adaptador de Objetos Portável (POA – Portable Object Adapter)

O Adaptador de Objetos Portável (POA-Portable Object Adapter) foi especificado pelo OMG para garantir o máximo de portabilidade entre os ORBs de diferentes vendedores. Ele tem um papel fundamental nessa dissertação, como veremos mais adiante.

#### 2.5.4.1. Referência de Objeto Inter-Operável (Interoperable Object Reference – IOR)

Todo servidor deve criar pelo menos um POA. Cada POA contém um conjunto de políticas que determinam o seu comportamento e um nome único, que o identifica no contexto do computador onde ele foi criado (Figura 7).

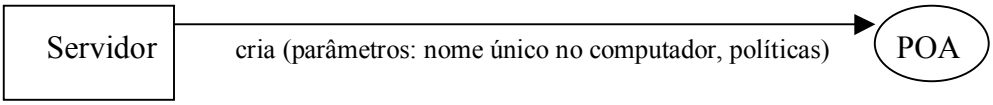


**Figura 7 – Criação do POA**

ORBs desenvolvidos por fabricantes diferentes são capazes de processar as IOR (utilizando os protocolos GIOP (General Inter-Object Protocol) / IIOP (Internet Inter-Orb Protocol)).

O Servidor solicita ao POA a criação de uma IOR, e a seguir exporta essa IOR para os seus clientes (arquivo texto, serviço de nomes, serviço de trader). De posse da IOR, os clientes podem enviar requisições para o Servidor (Figura 8).

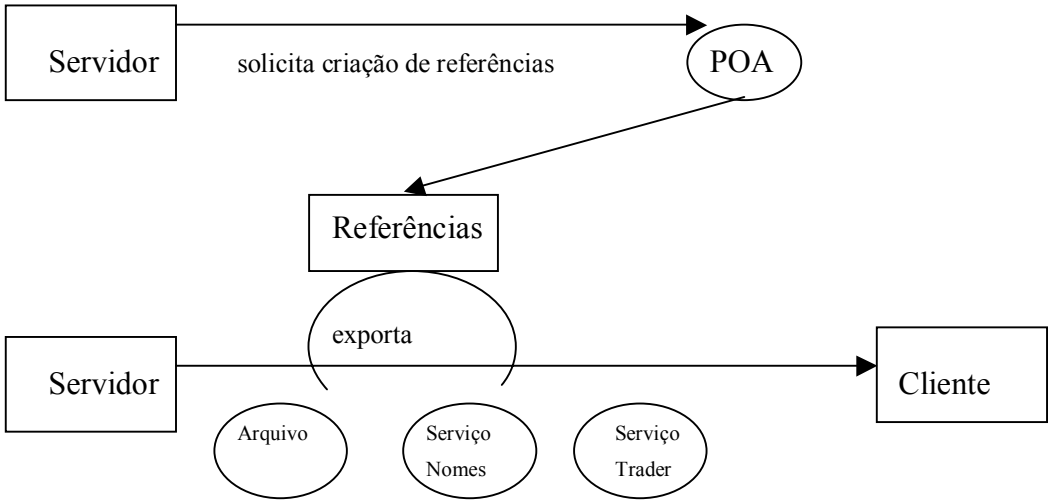


**Figura 8 – Criação de IOR**

Simplificando, a IOR contém:

- O endereço da máquina servidora: permite que o ORB identifique o computador correto para onde a requisição deve ser enviada e gerencia o roteamento apropriado das mensagens.
- O nome do POA que criou a referência para o objeto
- Um ObjectId: identifica o objeto, e é usado para rotear as requisições para um servant que irá atendê-las. Um servant pode atender requisições para vários objetos e utiliza o ObjectId para fazer a identificação/distinção entre eles.

A demultiplexação das requisições é feita conforme indicado na Figura 9.

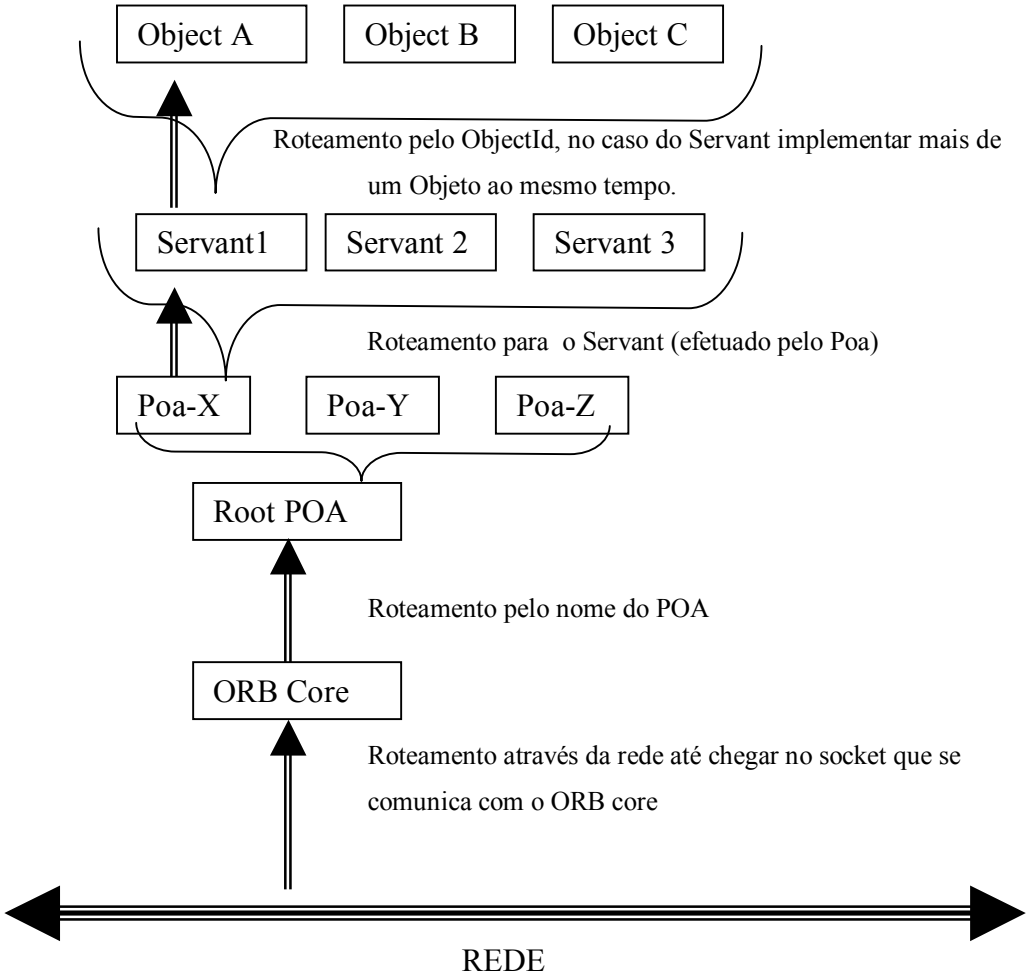


**Figura 9 – Demultiplexação da Referência**

O POA é um objeto como outro qualquer: é criado, tem uma referência, é destruído, é invocado. A única diferença é que ele é um objeto local, ou seja, a sua referência só faz sentido para o ORB dentro do servidor no qual o POA foi criado. Sua missão é conectar as requisições aos respectivos Servants em um computador específico. Desse modo, a implementação de um objeto pode ser vista como a combinação de um POA e de um Servant.

Os POAs não são persistentes. É responsabilidade da aplicação Servidora criar os POAs apropriados na sua inicialização.

Ao criar um POA é preciso definir qual o modelo de ativação que ele seguirá. Os modelos de ativação determinam o modo e momento de criação dos Servants. A aplicação pode criar os Servants ela mesma e cadastrá-los junto ao POA (ativação explícita, ativação

implícita), criar um Servant padrão que será chamado para processar todas as requisições que chegarem ao POA (ativação Default Servant), ou definir um Gerente de Servant (Servant Manager), que será invocado pelo POA sempre que for necessária a criação de um Servant para atender uma requisição (ativação sob-demanda).

#### 2.5.4.2. Políticas de Comportamento

Todo POA tem um conjunto de políticas que definem seu comportamento. As políticas controlam, entre outras funcionalidades, a geração das referências de objeto, a geração dos ObjectIds, o modelo de interação entre o POA e os servants (modelos de ativação) e o gerenciamento multi-thread e de transações.

Cada POA poderá ter políticas diferentes, conforme as necessidades da aplicação. Por exemplo, a aplicação pode criar dois POAs, um para lidar com objetos transientes, e outro para cuidar dos objetos persistentes.

Todas as aplicações têm pelo menos um POA, chamado rootPOA, cuja referência é disponibilizada pelo ORB. A aplicação pode criar POAs aninhados dentro do rootPOA, chamando rootPOA.create_POA(). As políticas de cada novo POA são definidas no momento da criação.

A especificação do POA descreve as seguintes políticas:

- **Política de Gerenciamento de Threads (Thread Policy)**
  - *single-threaded*: todas as requisições enviadas para os Servants pelo POA são serializadas
  - *multi-threaded*: o ORB determina/controla quais threads do POA estarão ativos
- **Política de Retenção do Servant (Servant Retention Policy)**
  - RETAIN: mantém a associação entre o servant e o objeto CORBA
  - NO_RETAIN: é estabelecida uma nova associação para cada requisição que chega para o objeto. Quando chega a requisição, o POA chama o Gerente de Servant indicado pela aplicação para a criação do servant.
- **Política de Processamento das Requisições (Requisition Processing Policy)**
  - USE_ACTIVE_OBJECT_MAP_ONLY - Consultar apenas o Mapa de Objetos Ativos (tabela que associa ObjectId com servant): é verificado se existe um servant

associado ao ObjectId do objeto CORBA destino. Se sim, a requisição é encaminhada para o objeto. Se não, o POA levanta a exceção CORBA::OBJECT_NOT_EXIST.

- USE_DEFAULT_SERVANT - Usar um Servant Default: a aplicação pode registrar um Servant default que será utilizado sempre que for enviada uma requisição para um objeto que não tem entrada no Mapa de Objetos Ativos.
- USE_SERVANT_MANAGER - Invocar um Gerente de Servant: a aplicação pode registrar um Gerente de Servant, que será invocado pelo POA para obter um Servant. O Gerente de Servant é capaz de encarnar ou ativar um servant e retorná-lo para o POA. Existem dois tipos de Gerentes de Servant: Ativador de Servant (no caso do POA ter a política de RETAIN), e Localizador de Servant (para a política NO_RETAIN).

- **Política de Ativação Implícita (Implicit Activation Policy)**
  - IMPLICIT_ACTIVATION: A aplicação cria o objeto e o servant correspondente para o objeto. A seguir o método _this() do servant é chamado, isso faz com que o servant seja implicitamente registrado junto ao RootPOA e uma referência para o objeto seja retornada.
- **Política de Unicidade de ObjectId (ObjectId Uniqueness Policy**)
  - UNIQUE_ID: Um servant pode encarnar um único objeto.
  - MULTIPLE_ID: Um servant pode encarnar vários objetos ao mesmo tempo, cada um com um objectId único.
- **Política de Ciclo de Vida (Lifespan Policy)**
  - determina se os objetos contidos pelo POA são transientes (TRANSIENT) ou persistentes (PERSISTENT). Quando o POA é desativado, as referências para os objetos transientes se tornam inválidas. Objetos persistentes continuam a existir mesmo que o POA a partir do qual eles foram criados seja desativado. Nesse caso, é implementado um mecanismo em que um novo POA é criado sob demanda caso a referência para o objeto persistente seja utilizada.
- **Política de Criação do ObjectId (Id Assignment Policy)**
  - define se os objectIds são definidos pelo POA (SYSTEM_ID) ou pela aplicação (USER_ID), no momento de cadastrar o objeto junto ao POA.

#### 2.5.4.3. Modelos de Ativação

Os modelos de ativação definem como o POA lida com os servants. Por exemplo, os Servants devem ser criados e cadastrados no POA antes que o POA comece a processar as requisições. Outra possibilidade, os Servants são criados e destruídos dinamicamente a medida que as requisições chegam.

Seguem os modelos de ativação do POA.

#### 2.5.4.4. Ativação Explícita:

O código da aplicação cria todos os servants e registra junto ao POA o servant que será utilizado para cada um dos objetos (Figura 10).

A associação Servant X Objeto é feita através do ObjectId. O POA possui uma tabela, chamada Mapa de Objetos Ativos, que contém em cada linha o par ObjectId, Servant.

O ObjectId pode ser gerado pela própria aplicação servidora, ou pelo POA. No primeiro caso, a aplicação utiliza a interface *activate_object_with_id* para registrar um par ObjectId,Servant junto ao POA. No segundo caso, a aplicação chama a interface *activate_object* para registrar um Servant, e um ObjectId único é automaticamente gerado pelo POA.

O que controla a geração do ObjectId é a Política de Criação do ObjectId do POA (ID Assignment Policy), que pode ter os valores USER_ID (no caso do ObjectId ser definido pela aplicação) ou SYSTEM_ID (quando o ObjectId é gerado automaticamente pelo POA).

A grande vantagem do modelo de Ativação Explícita é que o tempo de resposta às requisições é muito rápido, pois todos os servants estão prontos para fazer o processamento. A desvantagem, é o consumo de recursos de memória e processamento para manter os servants executando.

O modelo de Ativação Explícita é propício nos seguintes cenários:

- a aplicação tem poucos objetos (assim o consumo de memória é reduzido);
- os objetos devem estar disponíveis o tempo todo e com alta velocidade de processamento das requisições; e
- os objetos são ativados pela aplicação para efetuar uma tarefa específica e depois desativados (objetos temporários).

Essa idéia de objeto temporário e/ou com tempo de vida bem delimitado é representada pela Política de Ciclo de Vida do POA (Life Span Policy).

Os objetos com Política de Ciclo de Vida TRANSIENT têm a mesma duração do processo que o está criando. Quando o processo morre, o objeto é destruído e a referência para o objeto torna-se inválida. A vantagem é que a referência dos objetos TRANSIENT contém pouca informação de roteamento, uma vez que apenas um processo em uma máquina é capaz de atender às requisições. Caso o processo não exista mais, ou haja uma falha, não é necessário procurar por outro processo, basta marcar a referência como inválida e gerar uma exceção para o cliente.

Já os objetos com LifeSpanPolicy PERSISTENT terão um tempo de vida maior do que o processo que os criou. E, portanto, precisam que suas referências contenham mais informações para roteamento e localização do Servant apropriado.

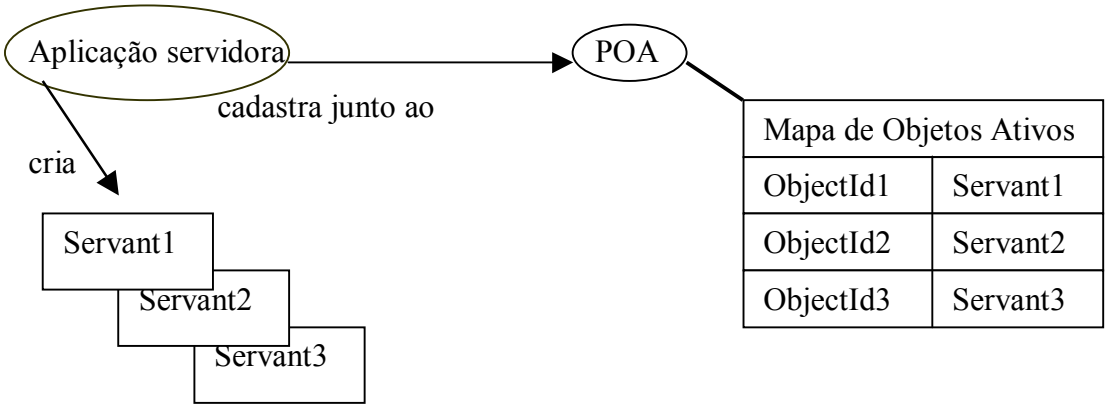


**Figura 10 – Ativação Explícita do POA**

Para funcionar exclusivamente sob o modelo de Ativação Explícita, o POA deve definir as políticas conforme a Tabela I.

**Tabela I – Políticas do POA para a Ativação Explícita**

| Política | Valor |
|---|---|
| Política de Criação do ObjectId (Id Assignment Policy) | • USER_ID: o ObjectId do servant é definido pela própria aplicação Server<br>• SYSTEM_ID: o ObjectId é gerado automaticamente pelo POA |
| Política de Ciclo de Vida | • TRANSIENT: os objetos têm o mesmo tempo de vida |

| | |
|---|---|
| (Life Span Policy) | que os processos que os criaram |
| Política de Processamento das Requisições (Requisition Processing Policy) | • USE_ACTIVE_OBJECT_MAP_ONLY: ao receber uma requisição, o POA busca apenas na tabela Mapa de Objetos Ativo pelo Servant correspondente ao ObjectId extraído da referência. |

O POA default (RootPOA) é inicializado com as políticas LifeSpan=TRANSIENT e IdAssignmentPolicy=SYSTEM_ID. É apropriado para aplicações que desejam utilizar o modelo de Ativação Explícita.

#### 2.5.4.5. Ativação Implícita:

A aplicação servidora cria um servant correspondente para um objeto. A seguir o método _this() do servant é chamado, isso faz com que o servant seja implicitamente registrado junto ao RootPOA e uma referência para o objeto seja retornada (Figura 11).

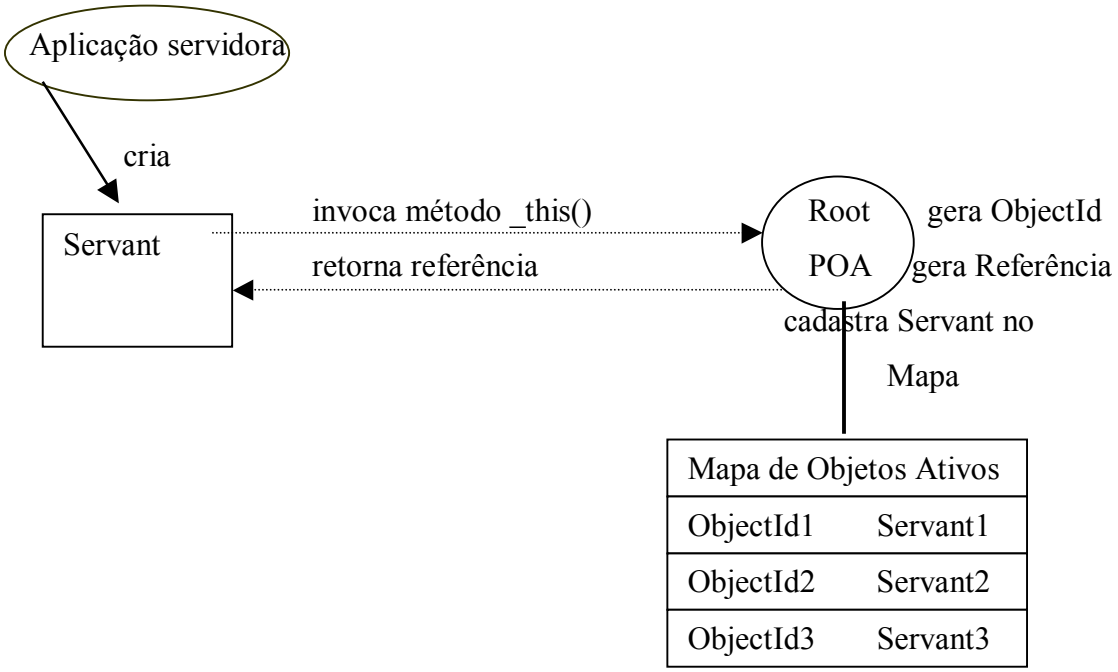


**Figura 11 – Ativação Implícita do POA**

Para funcionar sob o modelo de ativação Default Servant, o POA deve definir as políticas conforme a Tabela II.

**Tabela II - Políticas do POA para a Ativação Implícita**

| Política | Valor |
|---|---|
| Política de Ativação Implícita (Implicit Activation Policy) | • IMPLICIT_ACTIVATION: A aplicação cria o objeto e o servant correspondente para o objeto. A seguir o método _this() do servant é chamado, isso faz com que o servant seja implicitamente registrado junto ao RootPOA e uma referência para o objeto seja retornada. |

#### 2.5.4.6. Sob-Demanda:

O código da aplicação servidora registra junto ao POA um Gerente de Servant que será chamado sempre que uma requisição for enviada para um objeto inativo.

Ao ser invocado o Gerente de Servant cria um servant para atender à requisição enviada pelo POA. Caso o objeto já tenha sido destruído, o Gerente de Servant levanta a exceção CORBA::OBJECT_NOT_EXIST. Caso o Gerente de Servant deseje implementar um esquema de balanceamento de carga, ele pode gerar uma exceção ForwardRequisition, indicando que o POA deve enviar a requisição para ser processada por um outro objeto.

Após o processamento da requisição, o servant recém-criado poderá ser cadastrado pelo POA no Mapa de Objetos Ativos. Assim, quando vier uma próxima requisição para o objeto, esse mesmo servant será automaticamente utilizado para atendê-la. Esse comportamento é determinado pela Política de Retenção do Servant (Server Retention Policy) com valor RETAIN (Figura 12).

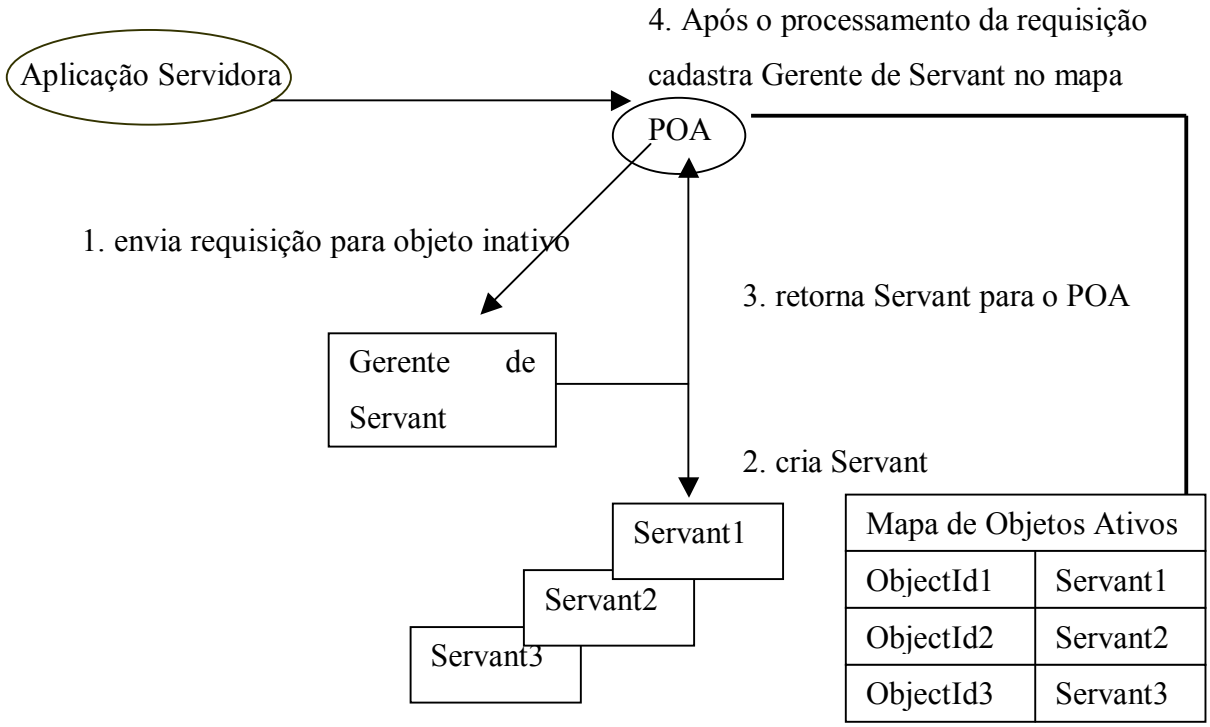


**Figura 12 –Ativação Sob-demanda (RETAIN) do POA**

O Gerente de Servant cadastrado junto a um POA com política RETAIN é chamado de Ativador de Servant (Servant Activator) e define duas operações:

- incarnate: cria um novo servant (se ele não existir) e atribui o seu estado inicial. O incarnate só é chamado se o servant não existir no Mapa de Objetos Ativos.
- etherealize: é chamado quando o objeto é desativado. O servant continua existindo, mas em estado inativo. Para ativá-lo basta chamar incarnate novamente.

Outra possibilidade é que após o processamento da requisição, o POA destrua o servant. Quando uma nova requisição for enviada para o objeto, será necessário solicitar ao Gerente de Servant a criação de um novo servant. Esse comportamento é determinado pela Política de Retenção do Servant (Server Retention Policy) com valor NON_RETAIN (Figura 13).

O Gerente de Servant registrado junto a um POA com política NON_RETAIN é chamado de Localizador de Servant (Servant Locator) e define duas operações:

- preinvoke: cria um novo servant e atribui o seu estado inicial.
- posinvoke: limpa o estado do servant (podendo até salvar o estado em base persistente), e destrói o servant.

Servants criados por um Gerente de Servant do tipo Localizador de Servant não são cadastrados no Mapa de Objetos Ativos do POA.

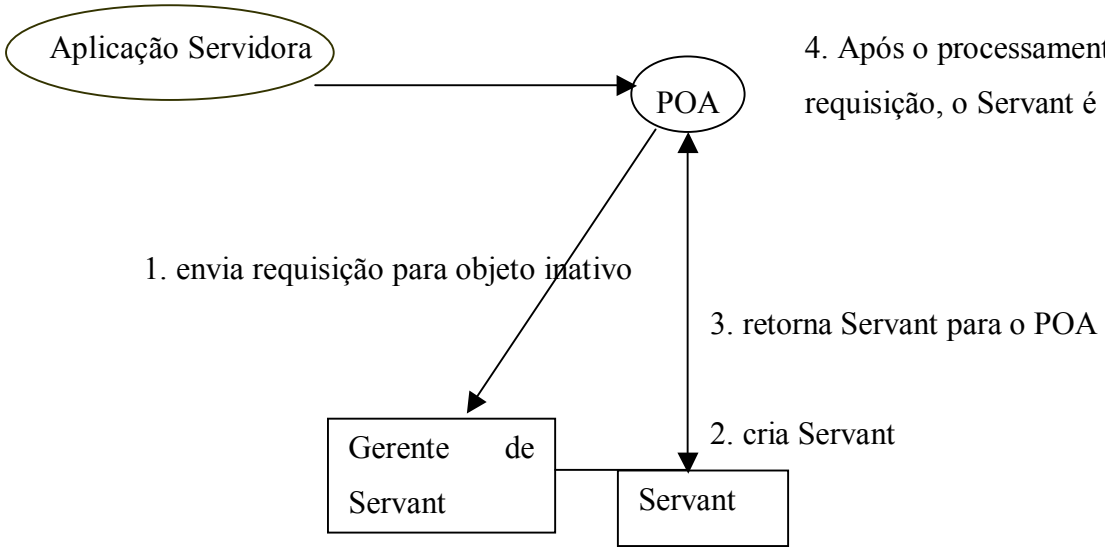


**Figura 13 - Ativação Sob-demanda (NON-RETAIN) do POA**

Para funcionar sob o modelo de ativação Sob-Demanda, o POA deve definir as políticas conforme a Tabela III.

**Tabela III - Políticas do POA para a Ativação Sob-Demanda**

| Política | Valor |
|---|---|
| Política de Processamento das Requisições (Requisition Processing Policy) | • SERVANT_MANAGER: um Gerente de Servant deve ser cadastrado junto ao POA para efetuar a criação dos servants |
| Política de Retenção do Servant (Servant Retention Policy) | • RETAIN: servants criados são guardados no Mapa de Objetos Ativos, e serão utilizados para processar as próximas requisições. O Gerente de Servant é chamado de Ativador de Servant.<br>• NON_RETAIN: um novo servant é criado para processar cada requisição e destruído ao fim do processamento. O Gerente de Servant é chamado de Localizador de Servant. |

#### 2.5.4.7. Default Servant:

A aplicação servidora registra um servant default que será utilizado sempre que uma requisição for enviada para o POA (Figura 14).

O Default Servant pode implementar uma única interface, e mesmo assim atender a vários objetos diferentes. Nesse caso, cada objeto é reconhecido pelo seu ObjectId.

O Default Servant tem muita utilidade para aplicações que utilizam a DSI (dynamic skeleton interface). Um único Default Servant pode atender múltiplas interfaces. Ele processa a requisição dinâmica, e cria um servant de "segundo nível" (específico para a interface) para atender a requisição.

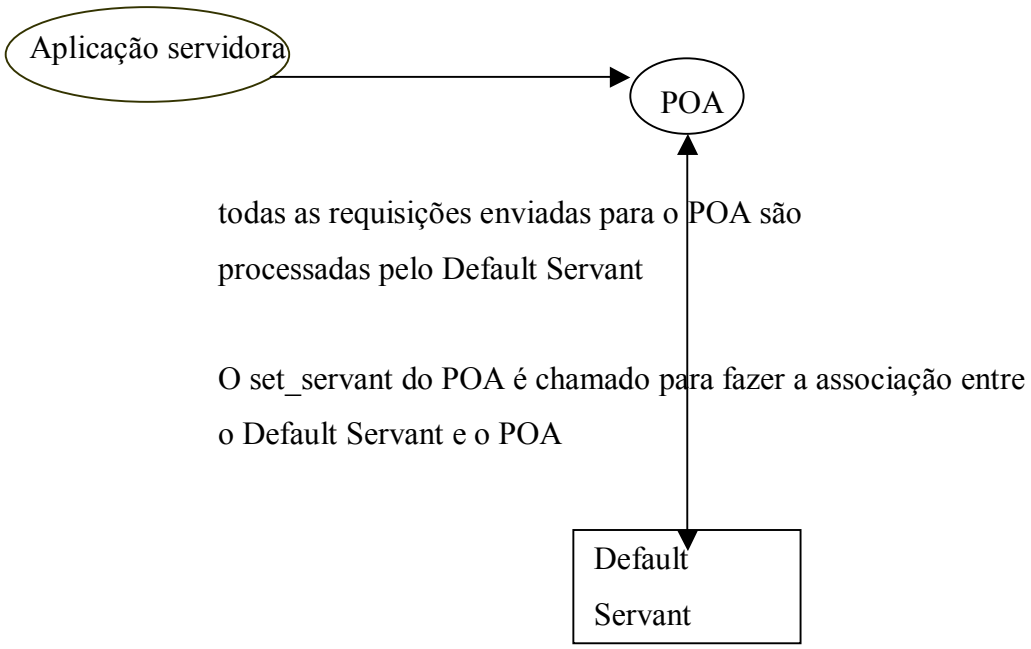


**Figura 14 – Ativação do Default Servant pelo POA**

Para funcionar sob o modelo de ativação Default Servant, o POA deve definir as políticas conforme a Tabela IV.

**Tabela IV - Políticas do POA para a Ativação Default Servant**

| Política | Valor |
|---|---|
| Política de Processamento das Requisições (Requisition Processing Policy) | • USE_DEFAULT_SERVANT: um mesmo Servant é utilizado para atender todas as requisições enviadas ao POA. |

# Capítulo 3
# Componentes de Software

## 3.1 Importância dos Componentes de Software

Um componente é uma unidade de software com interfaces e dependências de contexto especificadas explicitamente. Um componente pode ser instalado independentemente ou formar composições com outros componentes [14].

A tecnologia de componentes promete revolucionar o desenvolvimento de software[10].

O desenvolvedor de um componente segue um padrão ditado por uma dada tecnologia de componentes, implementando as interfaces requeridas para cada comportamento.

Os componentes são pedaços de software, que justamente por seguirem um padrão bem definido, se tornam facilmente reutilizáveis pelas aplicações. Acabam formando um framework, que possibilita o desenvolvimento rápido de aplicações complexas. Para isso, basta escolher um conjunto de componentes, configurar as suas propriedades e efetuar as conexões entre eles.

## 3.2 Principais Características dos Componentes de Software

A Figura 15 ilustra as principais características de um componente:

- **Propriedades**: atributos que são modificados para efeitos de configuração do componente.
- **Eventos**: os componentes de software podem se comunicar através da troca de eventos. Um componente interessado em receber eventos de um determinado tipo/categoria deve se cadastrar como um "consumidor" (subscriber). O componente que efetivamente emite o evento cadastra-se como "produtor" (publisher). Esses cadastros são efetivados no momento da instalação e ativação dos componentes.
- **Interfaces**: pontos de acesso aos métodos (operações) implementados pelo componente
- **Introspecção (Meta-Informação):** o componente deve ter a capacidade de responder consultas sobre as propriedades, eventos e interfaces que ele suporta.
- **Customização**: edição das propriedades do componente para efeitos de configuração.

- **Persistência**: empacotamento e distribuição dos componentes (devidamente configurados e conectados) pelos computadores (máquinas servidoras) apropriados. É usada como suporte à customização (edição das propriedades) e conexão entre os componentes (envio/recepção de eventos por exemplo).

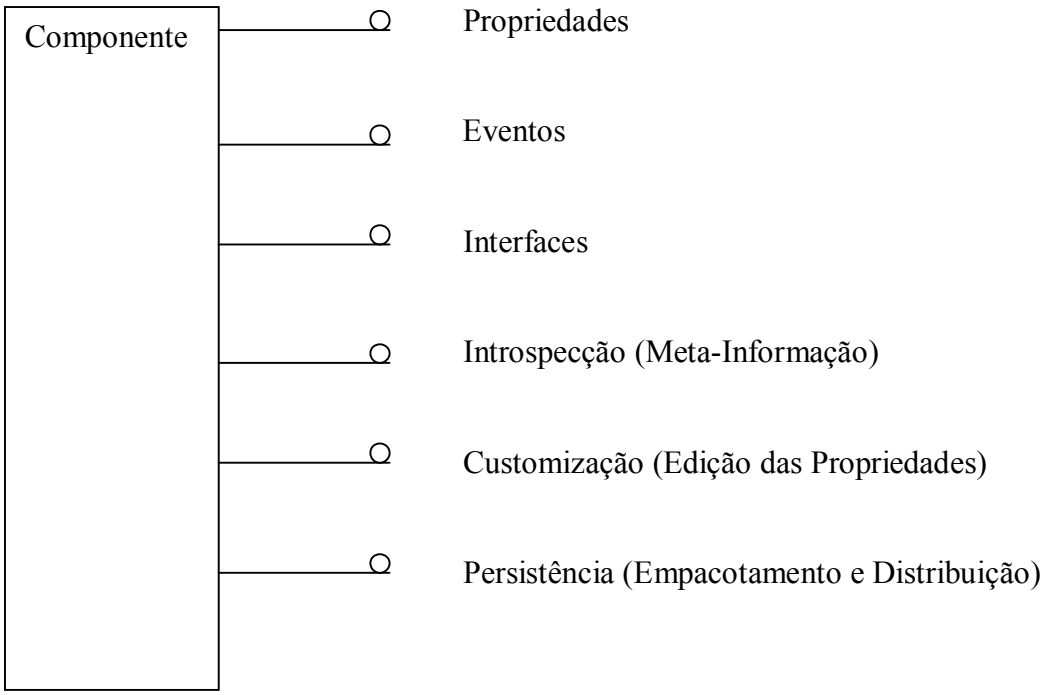


**Figura 15 – Características dos Componentes de Software**

## *3.3 Vantagens do Uso de Componentes de Software*

As aplicações baseadas em componentes são muito compactas. Em geral, elas contêm apenas o código relacionando a semântica de negócio (business) sendo implementada. Não é necessário codificar elementos de infra-estrutura e comunicação, pois estes já se encontram encapsulados no uso da tecnologia de componentes.

A integração/interação com outras aplicações baseadas em componentes é possível, uma vez que o próprio desenvolvimento baseado em componentes torna cada aplicação modular e com uma interface bem definida.

A escalabilidade é mais fácil de implementar e manter. É possível trocar um módulo completo de uma aplicação, por outro mais eficiente, sem modificar nenhuma linha de código.

## 3.4 Tecnologias de Componentes

Nas seções seguintes comentamos algumas das tecnologias de componentes que têm sido utilizadas ultimamente: JavaBeans e EnterpriseJavaBeans.

### 3.4.1. JavaBeans

JavaBeans [6] define um modelo de componentes de software para Java. Foi criado pela Sun Microsystems em 1996, e tem evoluído a cada ano.

O Java Bean é um componente de software reutilizável que pode ser manipulado visualmente através de uma ferramenta gráfica do tipo "builder".

O Bean expõe informações sobre os seus métodos, propriedades e eventos através do mecanismo de introspecção e das classes de suporte ao componente. As ferramentas de desenvolvimento do tipo "builder" podem mostrar essas informações para o desenvolvedor, permitindo que ele as modifique e combine os Beans para formar aplicações.

Os componentes Bean podem ser combinados para formar: outros componentes, applets (que são carregados em tempo de execução), aplicações Java stand-alone, ou servlets (usados para processar comandos e retornar páginas web).

A comunicação entre os JavaBeans é feita através do envio/recepção de eventos. O mecanismo de persistência possibilita serializar os JavaBeans.

Um dos principais objetivos do JavaBeans é a portabilidade. Cada Bean deve ser capaz de rodar em diversos ambientes. Além disso, é possível a implementação de pontes entre o modelo de componentes JavaBeans e outros modelos de componentes, como COM e ActiveX, facilitando a interoperabilidade.

### 3.4.2. Enterprise JavaBeans

JavaBeans é um modelo de componentes mais orientado para aplicações clientes (front-end). Em 1998 foi lançado o Enterprise JavaBeans [11], que adicionou várias funcionalidades para tornar o modelo JavaBeans mais apropriado para aplicações servidoras (back-end).

O EJB suporta dois modelos básicos para transações: bean de sessão e bean de entidade. O bean de sessão representa um objeto transiente, sem estado e com tempo de vida curto (é

criado pelo cliente e destruído após sua utilização). Já o bean de entidade representa objetos transacionais, com estado e tempo de vida longo.

O modelo EJB provê uma série de serviços, entre eles: gerenciamento de ciclo de vida, segurança, transações, persistência e gerenciamento de estado.

## *3.5 Componentes CORBA*

### 3.5.1. A especificação de Componentes CORBA

Em 1997, o OMG abriu uma requisição por propostas (Request for Proposals - RFP) solicitando propostas para um Modelo de Componentes em CORBA. Esse modelo deveria definir os componentes como uma extensão natural da CORBA, aproveitar as especificações CORBA existentes, e usar como base os modelos de componentes já existentes como JavaBeans.

A especificação de Componentes CORBA publicada em agosto de 1999 [24] procurou unificar as visões e soluções apresentadas por todas as submissões feitas em resposta à RFP. Várias das soluções propostas foram implementadas pelos vendedores e testadas pelo usuários. A escolha de quais propostas fariam parte dessa proposta final foi feita levando-se em conta as experiências e os resultados obtidos pelos vendedores e usuários. Esse processo garantiu provas de conceitos suficientes para fundamentar a proposta final; cujos pontos principais são:

- define extensões à IDL para definições dos componentes CORBA e das relações entre eles;
- introduz a CIDL (Component Implementation Definition Language), uma linguagem semelhante a IDL, que descreve as implementações dos componentes (seus estados, suas fábricas, entre outros);
- define extensões no Núcleo da CORBA para integrar o conceito de Componentes na OMA;
- define interfaces de navegação entre as várias interfaces de um Componente;
- define mecanismos para efetuar ligações entre os componentes;
- define o modelo de informação para o deployment dos componentes (usa XML);
- define o modelo de Container (entidade que gerencia o ciclo de vida dos componentes, em conjunto com o POA e o ORB);

- define políticas para suportar: transações CORBA, segurança e gerenciamento de estado persistente;
- define políticas para gerenciar o ciclo de vida de um servant (otimização da utilização dos recursos); e
- define integração com JavaBeans.

O modelo de Componentes para CORBA foi criado tendo como inspiração o JavaBeans, e prevê na sua especificação o modo de implementar uma ponte entre os dois modelos.

### 3.5.2. Modelo de Componentes CORBA (CCM – CORBA Component Model)

O CCM estende o modelo de objetos CORBA tradicional, definindo funcionalidades e serviços para a implementação, gerenciamento, configuração e deployment de componentes.

O CCM compreende as seguintes partes integradas e inter-relacionadas: Modelo Abstrato de Componente, *Framework* para Implementação de Componente, Modelo de Programação de *Container* para Componentes, Arquitetura do *Container* para Componentes, Integração com persistência, transações e eventos, Deployment e Empacotamento de Componentes, Integração com EJB 1.1 e Meta Modelo para Componentes. Nas seções seguintes mostraremos alguns detalhes das partes relevantes para a compreensão dessa dissertação. O *Container* é o *framework* onde o Componente será executado.

#### 3.5.2.1. Principais características dos Componentes CORBA

Um componente é formado por:

- **Facetas (Facets):** interfaces disponibilizadas pelo componente para interação com o cliente
- **Receptáculos (Receptacles):** pontos de conexão para o componente utilizar uma referência suprida por um agente externo;
- **Fonte de Eventos (Event Sources):** ponto de conexão que emite evento para um ou mais consumidores de eventos ou para um canal de eventos;

- **Sorvedouro de Eventos (Event Sinks):** ponto de conexão onde os eventos de um determinado tipo podem ser colocados (depositados, empurrados) para serem consumidos;
- **Atributos (Attributes):** valores que podem ser acessados/modificados através de operações próprias;
- **Chaves Primárias (Primary Keys):** valores exportados para os clientes que ajudam a identificar uma instância de componente em particular (opcional);
- **Interface Home (Home Interface):** interface padrão que provê operações de *factory* (responsável por criar uma nova instância do componente) e *finder* (responsável por localizar uma instância do componente já existente). Gerencia o ciclo de vida de um componente.

Na Figura 16 temos a ilustração de um componente.

#### 3.5.2.2. O Tipo Componente

O tipo componente será especificado em CIDL (Component Implementation Definition Language) que é uma IDL estendida, e armazenado no Repositório de Interfaces.

A definição do componente é na verdade uma especialização da definição de interface. De certa forma, é equivalente a permitir a definição de interfaces aninhadas.

Um componente tem:

- uma referência própria que corresponde à definição da interface do componente (chamada interface equivalente);
- referências disponibilizadas (provided references) para as interfaces disponibilizadas pelo componente (facetas).

Uma instância de um componente é identificada pela referência do componente. O ciclo de vida das interfaces disponibilizadas é o mesmo do componente que as contém. Os clientes podem navegar pelas interfaces disponibilizadas (facetas) e determinar se duas referências pertencem a mesma instância do componente.

O componente pode suportar herança de múltiplas interfaces, chamadas interfaces herdadas (supported interfaces), cujos métodos são acessados através da referência do componente.

A definição do componente suporta apenas uma única herança de um outro componente.

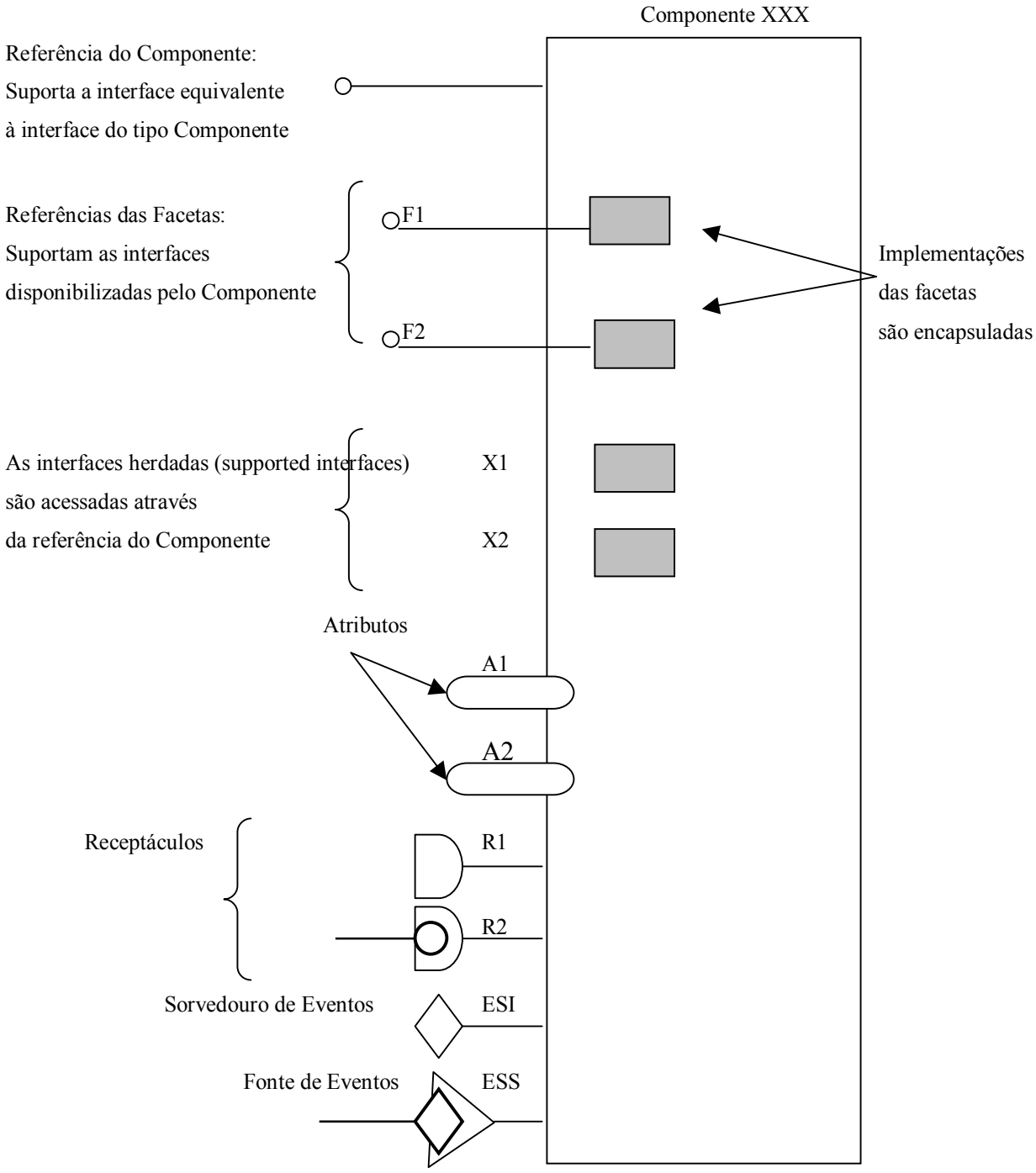


**Figura 16 – Ilustração de um Componente**

Exemplo de definição em IDL estendida (CIDL) do componente ilustrado na Figura 16 mostrada anteriormente é a seguinte:

```
interface X1, X2;             // declaração prévia (forward declaration)
component compBaseOne;
component XXX : compBaseOne supports X1, X2
      // definição da interface equivalente
      // e definição das "supported interfaces" A e B (interfaces herdadas)
{
      provides F1, F2;              // Facets (provided interfaces)
      uses InterfaceType R1, R2;    // Receptacles
      consumes EventType ESI;       // Event Sink
      publishes EventType ESS;      // Event Source outra notação possível é
                                    // emits EventType ESS;
      attribute AttributeType A1, A2;     // Attributes
}
```

Os componentes podem ser agrupados formando uma montagem (assembly) de componentes, como ilustrado na Figura 17.

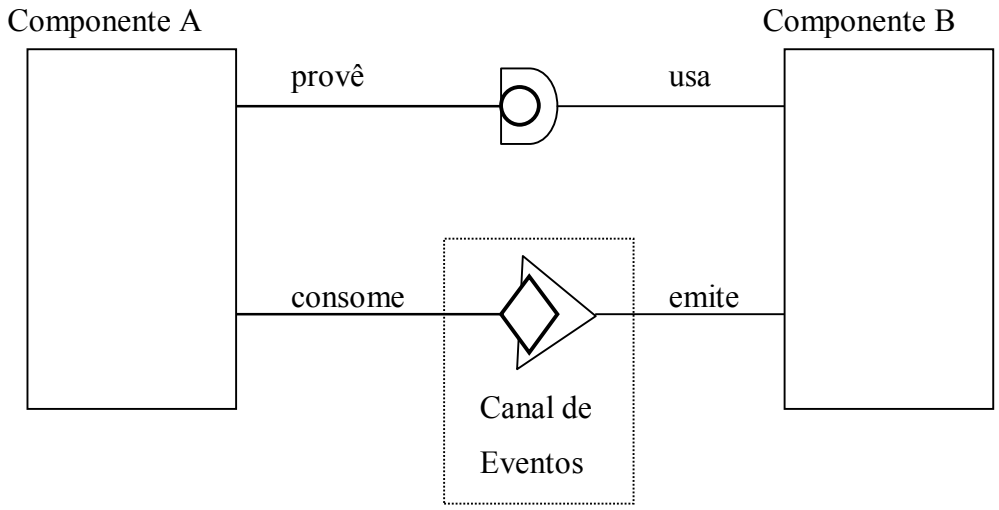


**Figura 17 – Montagem (Assembly) de Componentes**

O Modelo de Eventos do CCM é baseado no "Event Sources Push-Model", ou seja, as Fontes de Eventos enviam eventos (push) para o Canal de Eventos, que por sua vez repassam para os Sorvedouros de Eventos.

As cláusulas *publishes* e *emits* na definição das Fontes de Eventos de um componente indicam o tipo de Fonte. ***Publishes*** indica uma única fonte para vários sorvedouros (Figura 18).

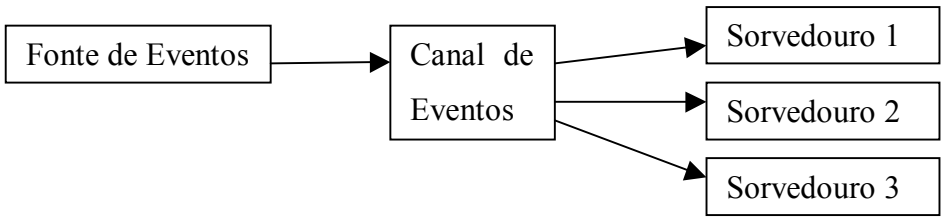


**Figura 18 – Fonte de Eventos do tipo Publisher**

***Emits***: indica uma única fonte para um único sorvedouro (Figura 19).

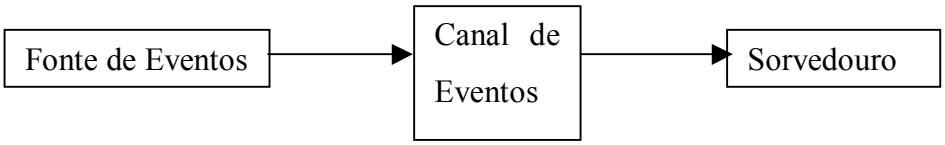


**Figura 19 – Fonte de Eventos do tipo Emitter**

Os componentes são executados dentro de uma entidade chamada Container, que é responsável por criar e gerenciar os canais de eventos.

#### 3.5.2.3. Modelo de Desenvolvimento e Empacotamento

Esse é um cenário típico dos passos a serem seguidos para a construção e utilização de componentes:

1. Análise

Nessa fase, o desenvolvedor deve coletar os requisitos da aplicação a ser desenvolvida, e decidir quais componentes serão necessários e que funcionalidades cada componente deve implementar.

2. Declaração e Implementação dos Componentes

O desenvolvedor deve escolher quais interfaces IDL farão parte do Componente tanto através de herança (interfaces herdadas) quanto através de agregação (interfaces disponibilizadas - facetas).

O desenvolvedor deve escrever a definição do Componente (atributos, facetas, receptáculos, fontes e sorvedouros de eventos) em CIDL.

O conjunto de definições IDL e CIDL é compilado, gerando:

- stubs (para o cliente, o servidor e a factory)
- a descrição do Componente (metadados em XML)
- o registro do Componente no Repositório de Interfaces.

Os stubs são preenchidos com código e compilados, gerando DLLs, bibliotecas compartilhadas ou Java bytecode, conforme o ambiente/linguagem de compilação (Figura 20).

3. Empacotamento dos Componentes

A compilação da CIDL gera um arquivo que descreve o componente em XML, com extensão .CCD (CORBA Component Descriptor), que contém:

- estrutura do componente (atributos, portas, eventos, etc...);
- repositoryId do componente (toda definição de componente é cadastrada no repositório de interfaces e toda instância de componentes no repositório de implementação);
- repositoryId e descrição de todas as interfaces que fazem parte do componente;
- a categoria do componente: serviço, sessão, processo, entidade (ver Seção 3.5.2.8);
- conjunto de arquivos de implementação do componente (um para cada plataforma). Alguns exemplos de arquivos de implementação são: uma DLL (Dynamic Linking Library), uma biblioteca compartilhada e um arquivo Java .class;
- arquivo com as propriedades do componente;
- repositoryId da interface Home do componente;

- arquivo com propriedade da interface Home do componente.

O arquivo Descrição do Componente é colocado em um arquivo (formato ZIP, extensão .CAR – Component Archive File).

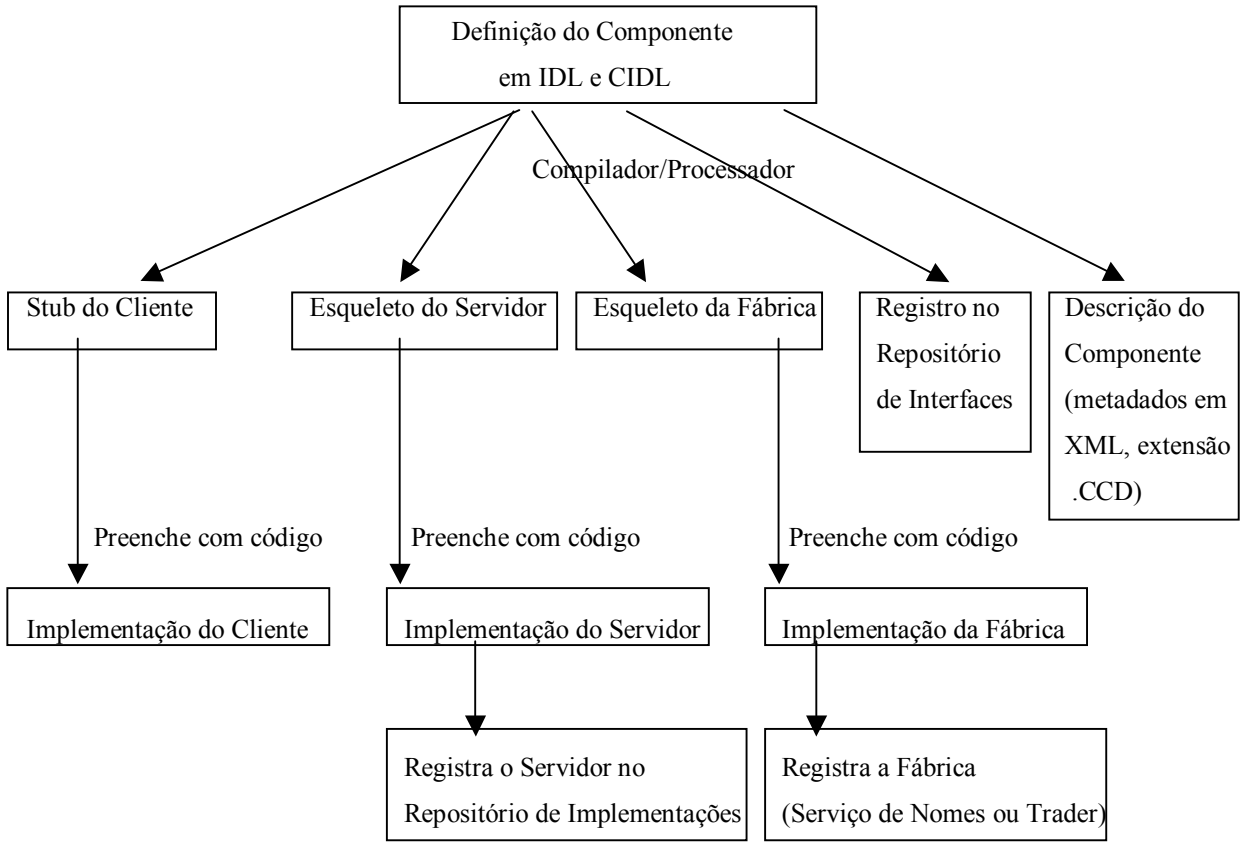


**Figura 20 - Declaração e Implementação dos Componentes**

4. Montagem dos Componentes

A montagem (assembly) é a descrição de como customizar e relacionar um dado conjunto de componentes.

Um componente que usa uma interface deve ser conectado ao componente que provê (implementa) essa interface. De modo análogo, um componente que consome um evento deve ser conectado ao componente que emite esse evento.

O arquivo de junção de componentes (Component Assembly Descriptor, extensão CAD) descreve em XML como as portas dos componentes (facetas, receptáculos, fontes e sorvedouros de eventos) são conectadas.

Geralmente, utiliza-se uma ferramenta visual do tipo "builder" para representar graficamente os componentes disponíveis, e:

- customizar as propriedades de cada componente;
- conectar os componentes entre si;
- particionar os componentes, ou seja definir conjuntos de componentes (partitions) que devem ser instalados juntos em um mesmo processo ou computador.

O resultado é um arquivo de Component Assembly (CAD) no formato comprimido (ZIP).

A seguir, cria-se um arquivo descritor da montagem de componentes, chamado Assembly Archive (extensão .AAR) que contém:

- o arquivo descritor de junção de componentes (.CAD);
- o conjunto de arquivos descritores de componentes (.CCD); e
- o conjunto de arquivos com propriedades de componentes (.CPF), se necessário.

5. Deployment dos Componentes

A especificação de Componentes CORBA deixa em aberto como será feito o deployment/instalação/distribuição dos componentes em um conjunto de computadores de uma rede (ambiente distribuído).

Cada vendedor deve especificar e implementar a sua própria ferramenta de deployment/instalação/distribuição.

A ferramenta deve ser capaz de lidar tanto com componentes individuais quanto com montagens de componentes. Tem como entrada um conjunto de arquivos com componentes e montagens de componentes, e a entrada do usuário indicando em quais computadores os componentes serão instalados. Deve instalar os componentes, as interfaces homes dos componentes e efetuar as conexões entre os componentes. O resultado deve ser componentes e montagens de componentes instalados e disponíveis para o uso.

O Componente executa em um Servidor de Components, que é um processo *standalone* que carrega a DLL associada à implementação do Componente.

#### 3.5.2.4. A Declaração do Componente em IDL

A interface IDL que descreve um componente contém: um cabeçalho, um corpo, e a declaração da interface de instanciação (Home interface).

O cabeçalho inclui:

- um nome que identifica o componente dentro do espaço de nomes;
- persistência: o estado de persistência do componente;
- categoria do componente (serviço, sessão, processo, entidade);
- herança: simples, de um único componente; e
- interfaces suportadas (herança).

Exemplos de cabeçalho de componente:

```
component Dog process { ... }
component Dog  : Animal { ... }
component Dog storedAs DogType process : Animal { ... }
```

O Corpo do componente inclui declarações de:

- constantes
- tipos
- exceções
- atributos
- operações
- interfaces disponibilizadas (facetas)
- interfaces utilizadas (receptáculos)
- eventos emitidos (fonte de eventos)
- eventos consumidos (sorvedouro de eventos)

Todas as funcionalidades dos Componentes descritas em IDL estendida têm as suas formas equivalentes na IDL simples (IDL da CORBA 2.3).

Como foi explicado, a definição de componente é um meta-tipo da definição de interface. Cada componente tem a sua "**interface equivalente**". Nos mapeamentos para linguagens de programação, os componentes são referências para objetos que implementam a interface equivalente determinada pela definição do componente.

Para ilustrar esse conceito de **IDL equivalente**, daremos exemplos de partes da definição de um componente (em IDL estendida) com o correspondente mapeamento para interfaces equivalentes (em IDL simples) conforme mostra a Tabela V.

**Tabela V – Mapeamento de IDL Estendida para IDL Simples**

| IDL Estendida | IDL Simples |
|---|---|
| `component component_name { ... }` | `interface component_name : CORBAComponent :: ComponetBase { ... }` |
| `component component_name : base_name supports interface_name_a { ... }` | `interface component_name : base_name, interface_name_a { ... }` |
| `provides interface_type MYname;` | `interface_type provide_MYname();` |
| `uses interface_type MYname;` | `void connect_MYname (in interface_type conn) raises (CORBAComponent::AlreadyConnected, CORBAComponent::InvalidConnection);`<br><br>`interface_type disconnect_MYname();`<br><br>`interface_type get_connected_MYname();` |
| `emits event_type MYname;` | `CORBAComponent::ObjectCookie add_listener_MYname (in CORBA::Component::EventListener lst);` |

| | |
|---|---|
| | `CORBA::Component::EventListener remove_listener_MYname (in CORBAComponent::ObjectCookie);`<br><br>`CORBA::Component::EventListeners get_listeners_MYname();` |
| `consumes event_type MYname;` | `CORBA::Component::EventListener get_consumer_MYname();` |

Em tempo de execução, o CCM gerencia as seguintes funções para os componentes:

- navegação
- identidade (através da chave primária)
- ativação/desativação
- gerenciamento do estado persistente
- criação/destruição

Fazendo declarações em CIDL os esqueletos necessários para disponibilizar as funcionalidades acima são automaticamente gerados.

#### 3.5.2.5. O Repositório de Interfaces para suportar os Componentes

Os programas podem tentar acessar o repositório de interfaces a qualquer momento chamando a operação *get_interface_def* a partir da referência para um objeto.

O Repositório de Interfaces é constituído por uma série de *interface repository objects* (os chamados Objetos do IR), que podem ser dos seguintes tipos:

**Repository**: é o módulo de nível mais alto do repositório (a raiz do espaço de nomes).

**ModuleDef**: agrupamento lógico de interfaces, tipos, valores, entre outros. Pode conter outros módulos.

**InterfaceDef**: definição de uma interface, contém uma lista de constantes, tipos, exceções, operações e atributos.

**ComponentDef**: definição de um componente, contém uma lista de interfaces (provides, uses, supports) e eventos (publishes, emits, consume) e atributos.

**HomeDef**: definição de um *home* de componente, contém uma lista de constantes, tipos, exceções, operações, atributos, *factories* e *finders*.

**ValueDef**: definição de um tipo *value*. Esse tipo define uma estrutura de dados com funcionalidades de leitura e modificação do seu conteúdo, ou seja, é uma *struct* com métodos *get*() e *set*(). O tipo *value* foi definido a partir da versão 2.3 da CORBA, e é usado para passar parâmetros por valor, pois o *value* não é um objeto CORBA, não tem uma referência, é apenas uma estrutura de dados.

**ValueBoxDef**: definição de um tipo *value box*. Semelhante ao tipo *value*. A diferença é que provê funcionalidades somente de leitura do seu conteúdo (possui apenas o método *get()*). Além disso, um parâmetro do tipo *value box* pode receber o valor NULL, enquanto que os parâmetros do tipo *value* não podem ser nulos.

**ValueMemberDef**: definição de um membro do tipo *value def*.

**AtributeDef**: definição de um atributo de uma interface, tipo *value*, *home* ou componente.

**OperationDef**: definição de uma operação de uma interface, tipo *value*, ou *home* e a lista de parâmetros e exceções levantadas pela operação.

**FactoryDef**: definição de uma fábrica, é uma operação que é utilizada especificamente para criar novas instâncias de objetos dentro de um *home*.

**FinderDef**: definição de um *finder*, é uma operação utilizada para descobrir instâncias dentro de um *home*.

**PrimaryKeyDef**: definição de uma chave, faz a associação entre um tipo *value* que contém uma chave e um *HomeDef*.

**TypedefDef:** interface base para a definição de tipos nomeados que não são interfaces nem tipos *value*.

**ConstantDef:** definição de uma constante nomeada.

**ExceptionDef:** definição de uma exceção que pode ser levantada por uma operação.

**ProvidesDef:** definição de uma interface que é disponibilizada por um componente.

**UsesDef:** definição de uma interface que é utilizada por um componente.

**EmitsDef:** definição de eventos que são emitidos por um componente.

**PublishesDef:** definição de eventos que são publicados por um componente.

**ConsumesDef:** definição de eventos que são consumidos por um componente.

Uma implementação de CORBA pode manter atributos adicionais que facilitem o gerenciamento do repositório, ou adicionar registros com informações proprietárias sobre as interfaces. As implementações que precisam estender o repositório devem fazer isso derivando novas definições a partir das definições de IR existentes.

O **ComponentRepository** é um tipo que deriva do **Repository** e é utilizado como o *container* de nível mais alto nível para os Objetos do IR que armazenam as informações relacionadas aos componentes.

#### 3.5.2.5.1. Estrutura Geral:

O Repositório de Componentes (*ComponentRepository*) pode armazenar definições de Componentes (*ComponentDef*) e de *Homes* de Componentes (*HomeDef*).

A Figura 21 mostra sucintamente a estrutura geral do Repositório de Componentes. No Apêndice A estão as definições completas do *ComponentRepository*, *ComponentDef* e *HomeDef*.

Existem quatro maneiras de localizar um componente no Repositório de Componentes:

1. Obter o objeto *ComponentDef* diretamente do ORB
2. Navegar pelo espaço de nomes do Repositório usando uma seqüência de nomes que dá a localização do componente
3. Localizar o *ComponentDef* que corresponde a um identificador dentro do repositório
4. Obter o *ComponentDef* a partir de um *HomeDef*

Existem três maneiras de localizar um *Home* no Repositório de Componentes:

1. Obter o objeto *HomeDef* diretamente do ORB
2. Navegar pelo espaço de nomes do Repositório usando uma seqüência de nomes que dá a localização do *home.*
3. Localizar o *HomeDef* que corresponde a um identificador dentro do repositório

É importante notar que dado um *ComponentDef* não é possível obter o *HomeDef* correspondente, pois pode haver vários *homes*.

Dada uma referência para um componente é possível obter o *ComponentDef* através da operação CCMObject::get_component_def.

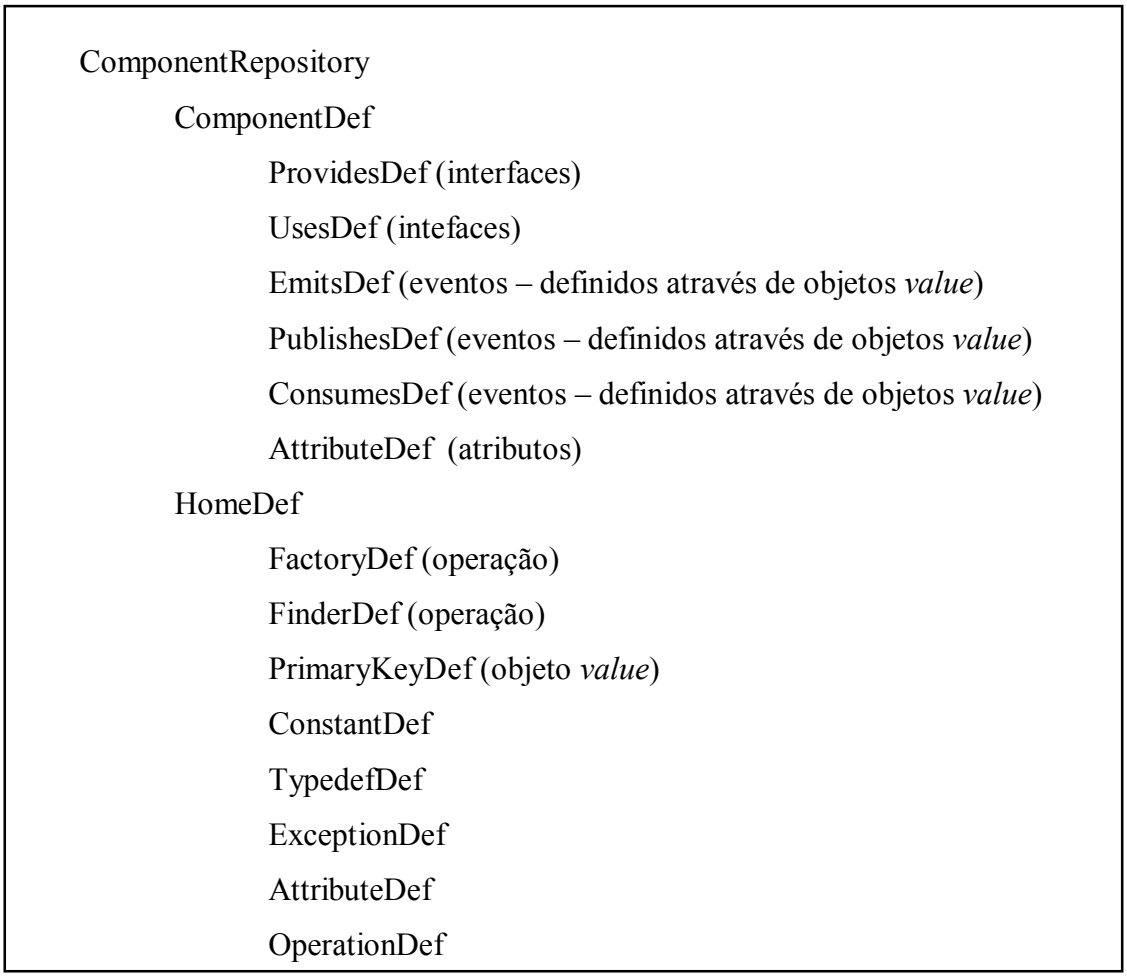


**Figura 21 – Estrutura Geral do Repositório de Componentes**

#### 3.5.2.6. Modelo de Programação do *Container*

O *container* é um *framework* que provê um ambiente de execução para os componentes, principalmente com relação aos serviços de transações, segurança, eventos e persistência em tempo de execução.

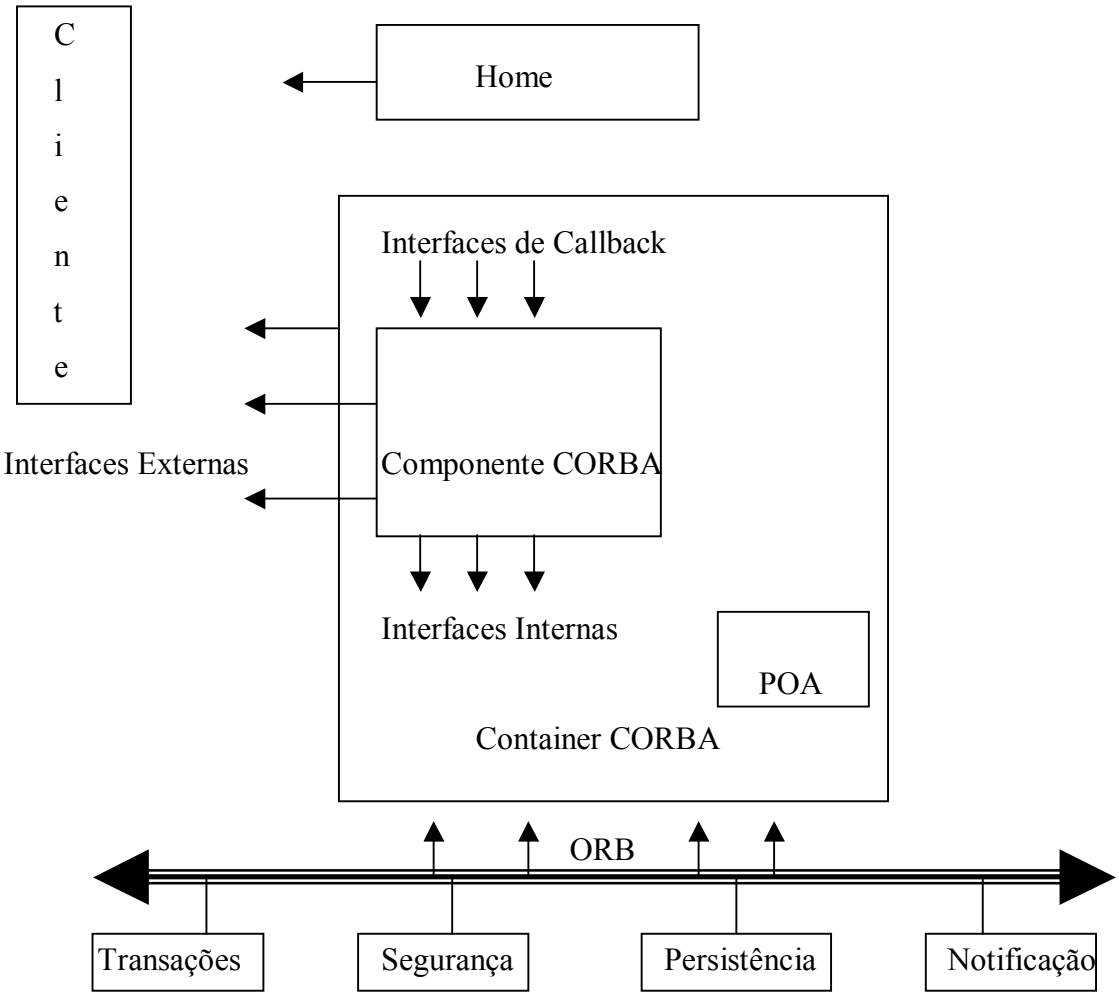


**Figura 22 – Modelo de Programação do Container**

O Container exerce as seguintes funções:

- todas as instâncias de um componente são criadas pelo seu *container*;
- o cliente tem acesso às interfaces disponibilizadas pelo componente através do *container;*
- o *container* provê para o componente serviços padronizados (transação, segurança, persistência, notificação);
- o *container* tem três tipos de interfaces (APIs – ver Figura 22):

  ⇒ externas: são as interfaces CORBA disponibilizadas para o cliente;

  ⇒ internas: são aquelas que implementam serviços para os componentes;

⇒ *callbacks*: utilizadas para gerenciar a ativação/desativação (*activation*/*passivation*) dos componentes. Todo componente deve cadastrar algumas funções de *callback* que serão utilizadas pelo *container*, como por exemplo: set_context, activate e passivate.

3.5.2.6.1. Tipos de Interfaces Externas

Representam o contrato entre o Componente e o cliente do Componente. Podem ser de dois tipos: *home* (interface para criação do componente) e aplicação (interface para a implementação do componente propriamente dita).

Do ponto de vista do cliente, existem duas formas de interação com as interfaces *home*:

- *factory*: utilizadas para criar novas instâncias de componentes;
- *finder*: utilizadas para buscar instâncias de componentes já existentes. Neste caso a interface *home* deve ter um parâmetro do tipo chave (chave primária).

3.5.2.6.2. Tipo de API do *Container* (Tipos de Interfaces Internas)

Há dois tipos, conforme a durabilidade da referência para o componente:

- Sessão: lida com componentes cujas referências são transientes;
- Entidade: lida com componentes cujas referências são persistentes.

### 3.5.2.7. Modelo de Uso CORBA

O modelo de uso CORBA especifica como é a interação entre o *Container*, o POA e os serviços CORBA:

- Sem estado: a referência para o componente é transiente. O POA redireciona a requisição para qualquer servant do tipo apropriado (escolhido de um pool de servants disponíveis).
- Conversacional: a referência para o componente é transiente. O POA associa um servant específico para cada referência de componente. Isso faz com que o cliente sempre "converse" com o mesmo servant enquanto a referência for válida.
- Durável: a referência para o componente é persistente. O POA associa um servant específico para cada referência de componente.

### 3.5.2.8. Categorias de Componentes

Cada categoria de componente define uma combinação entre o Modelo de uso CORBA e o tipo de API do *Container* (Tabela VI):

**Tabela VI – Categorias de Componentes**

| Categoria do Componente | Modelo de Uso CORBA | Tipo de API do *Container* | Referência para o Componente | Interface Home (com chave primária) |
|---|---|---|---|---|
| Serviço | Sem estado | Sessão | Transiente | não |
| Sessão | Conversacional | Sessão | Transiente | não |
| Processo | Durável | Entidade | Persistente | não |
| Entidade | Durável | Entidade | Persistente | sim |

#### 3.5.2.8.1. Componente de Serviço:

- Não tem estado;
- Não tem identidade;
- Qualquer instância do componente possui o mesmo comportamento; e
- Útil para componentes que executem tarefas corriqueiras e/ou repetitivas, e que sejam chamados muitas vezes. Como podem ser alocados a partir de um pool de instâncias equivalentes e reutilizáveis, contribuem para maximizar o desempenho.

#### 3.5.2.8.2. Componente de Sessão:

- Tem estado transiente;
- Tem identidade não-persistente; e
- Mantém um contexto que só faz sentido naquela sessão de interação entre o cliente e o servidor. Exemplo: indicação da posição em uma seqüência.

#### 3.5.2.8.3. Componente de Processo

- Tem estado persistente;
- Tem identidade persistente; e

- Geralmente utilizado para implementar processos com começo-meio-e-fim. O cliente pode interagir com o componente repetidas vezes para executar tarefas, mas uma vez que o processo como um todo se complete, a referência para aquele componente se torna inválida. Exemplo: componente para efetuar a matrícula de um aluno ingressante, após a submissão dos dados digitados, o componente deixa de existir.

#### 3.5.2.8.4. Componente de Entidade

- Tem estado persistente;
- Tem identidade persistente;
- O cliente acessa o componente através de uma chave (chave primária); e
- Comportamento semelhante aos componentes de Processo. Exemplo: componente para efetuar a alteração da matrícula de um aluno (a chave é o registro acadêmico do aluno).

### 3.5.2.9. Políticas de gerenciamento do Ciclo de Vida do Servant do Componente

As políticas de gerenciamento de Ciclo de Vida podem ser as seguintes:

- Por método: para cada requisição, o *container* escolhe um componente de um *pool* de instâncias reutilizáveis, ativa esse componente, redireciona a requisição a ser processada, e desativa (*passivate*) o componente ao final do processamento;
- Por transação: o *container* ativa o componente na primeira instrução de uma transação e desativa ao final da transação;
- Por componente: o *container* ativa o componente ao chegar a primeira requisição e o deixa ativo até que a própria implementação do componente solicite a desativação (*passivate*); e
- Por container: o *container* ativa o componente ao chegar a primeira requisição e o deixa ativo até que seja necessário desativá-lo (*passivate*).

A relação entre a categoria de componente e o gerenciamento do ciclo de vida do servant está na Tabela VII.

**Tabela VII – Categoria de componente X Gerenciamento do ciclo de vida do Servant**

| Modelo de Uso CORBA | Tipo de API do *Container* | Políticas válidas de Gerenciamento de Ciclo de Vida do Servant |
|---|---|---|
| Sem estado | Sessão | método |
| Conversacional | Sessão | método, transação, componente, *container* |
| Durável | Entidade | método, transação, componente, *container* |

#### 3.5.2.10. Arquitetura do Container

Na especificação é indicado um possível projeto (o mais simples) para a implementação de um *container*.

O processo de instalação deve criar um Gerente de *Container* (Figura 23).

O Gerente de *Container* cria os demais *containers*, um para cada categoria de componente, conforme as informações contidas no pacote de instalação do componente ou montagem de componentes.

Cada *container* inclui um POA especializado, responsável por criar referências e gerenciar os servants para os componentes do *container*.

Existem várias possibilidades de políticas e modelos de ativação para os POAs. Na especificação escolheu-se a mais simples de todas, baseada em ativação sob-demanda com Gerente de Servant do tipo Localizador de Servant.

Embora o Container seja o *framework* onde o Componente será executado, há uma série de itens a serem providenciados para que esse *framework* funcione de acordo. As responsabilidades de providenciar esses itens devem ser divididas entre a própria implementação do Container e o Processo de Instalação/Deployment dos Componentes.

O exemplo de arquitetura de *Container* apresentado na especificação é muito simples. O propósito é exemplificar quais atividades/funcionalidades devem ser suportadas pelos *containers* para que eles sejam considerados em conformidade. Várias arquiteturas complexas são possíveis, principalmente porque o processo de Deployment (que cria os *containers*) é deixado em aberto na especificação.

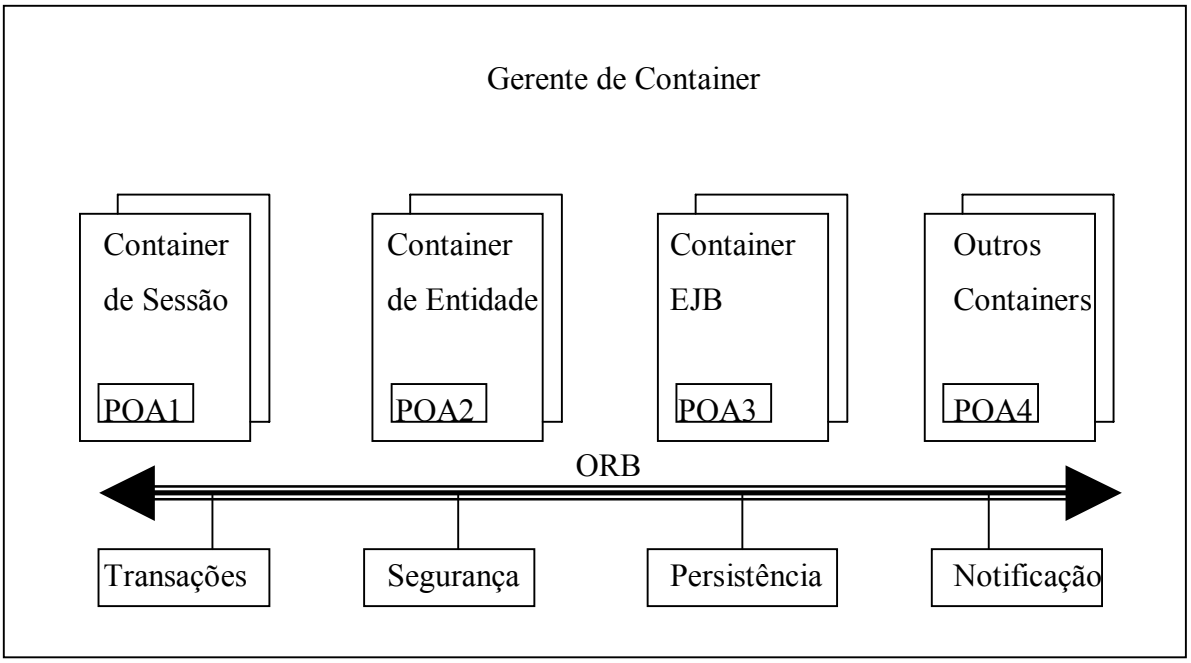


**Figura 23 – Arquitetura de um Gerente de Container**

#### 3.5.2.11. A visão do Cliente

O cliente vê dois tipos de interfaces: interfaces de aplicação (disponibilizadas pelo componente) e interfaces de localização (para obter referências para as interfaces de aplicação).

Alguns clientes estão cientes de que estão usando um componente (*components-aware*) e, por isso, podem utilizar recursos específicos dos componentes como por exemplo, a navegação por múltiplas interfaces, ou a busca por referências de componentes via *homes*, como exemplificado na Figura 24. O Cliente interage com o ORB para obter a referência para o Home Finder. A seguir interage com o Home Finder para obter a referência do Home capaz de criar o componente Z (Home Z). Por fim, interage com o Home Z para obter a referência para o componente Z.

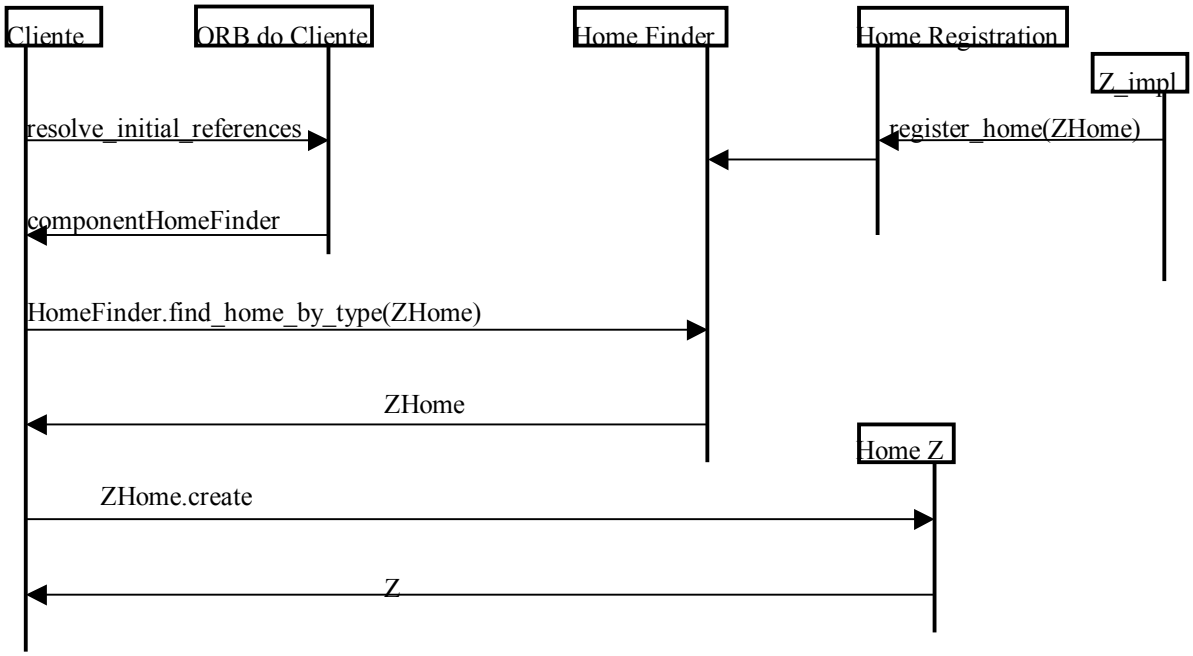


**Figura 24 – Fluxo de eventos para criação do componente Z por um cliente ciente do uso de componentes**

Outros clientes (*components-unaware*), não sabem se estão fazendo chamadas a interfaces comuns, ou interfaces disponibilizadas por componentes. Daí, eles só usam os recursos padrões do CORBA, como por exemplo, busca pelo nome das interfaces em um serviço de Nomes ou de Trader, como na Figura 25 a seguir. O Cliente interage com o ORB para obter a referência do Serviço de Nomes. A seguir, pergunta para o serviço de Nomes a referência do Home capaz de criar o componente Z (Home Z). Por fim, interage com o Home Z para obter a referência para o componente Z.

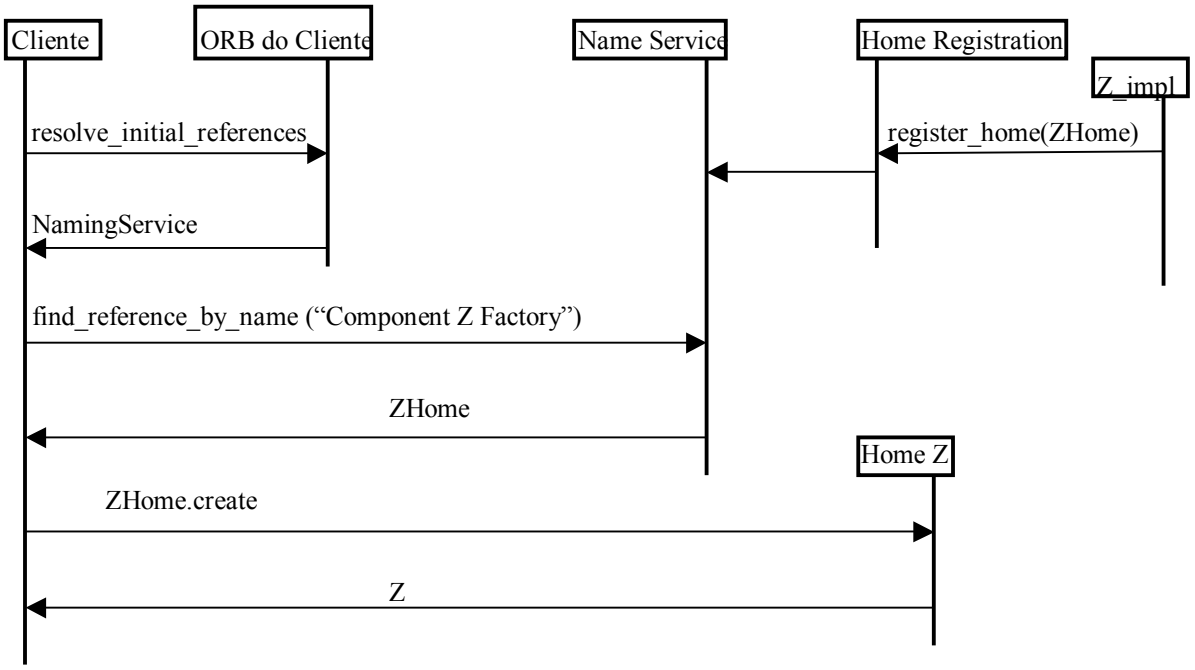


**Figura 25 - Fluxo de eventos para criação do componente Z por um cliente não-ciente do uso de componentes**

### 3.5.3. Deployment de Componentes CORBA

Os componentes, os *homes* dos componentes e as montagens de componentes (grupo de componentes interconectados) são instalados e ativados nas máquinas apropriadas utilizando-se uma ferramenta/aplicação de deployment.

O propósito do deployment é instalar e "costurar" uma topologia lógica de componentes em um ambiente computacional físico.

As informações necessárias para o deployment estão contidas no arquivo de montagem de componentes AAR (Assembly Archive) conforme descrito anteriormente no Modelo de Empacotamento (ver Seção 3.5.2.3 itens 3 e 4).

#### 3.5.3.1. O Processo de Deployment

Os passos principais do processo de Deployment são:

1. Identificar em quais máquinas os componentes devem ser instalados. Geralmente essa informação é obtida através da interação entre o usuário e a ferramenta de deployment;

2. Instalar as implementações apropriadas dos componentes nas máquinas especificadas (escolher a implementação conforme a plataforma de cada máquina);
3. Instanciar os componentes e os Homes dos componentes nas máquinas especificadas no passo 1; e
4. Conectar as instâncias dos componentes conforme especificado no arquivo de montagem.

#### 3.5.3.2. Arquitetura de Deployment

O modelo CCM descreve a arquitetura de deployment através de um diagrama de classes que requer as interfaces `ComponentInstallation`, `AssemblyFactory` e `Assembly` e supõe o uso de uma aplicação de deployment. O `ComponentInstallation` cuida da instalação do componente, ou seja, da distribuição do software que implementa o componente pelas máquinas apropriadas. O `AssemblyFactory` é responsável por obter todas as informações necessárias (tanto do sistema, quanto do usuário) para a criação do Assembly (montagem de componentes). O `Assembly` efetivamente instancia e ativa as instâncias dos componentes que fazem parte da montagem, deixando-as prontas para atender requisições. Na Figura 26 apresentamos as demais interfaces apenas para fins ilustrativos.

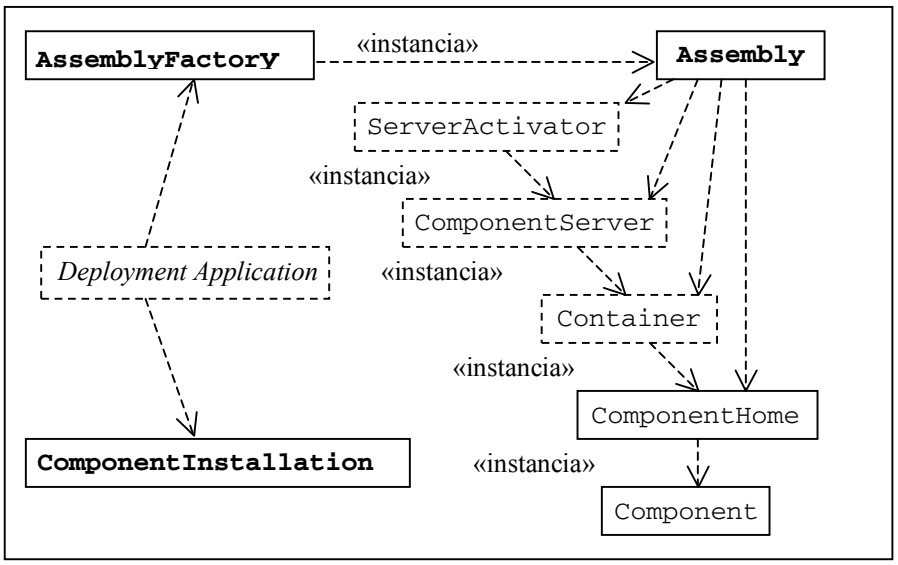


**Figura 26 – Diagrama de classes que mostra as dependências entre os participantes da arquitetura de deployment proposta pela CCM**

#### 3.5.3.3. Cenário de Deployment

Considerando-se o diagrama de classes apresentado na seção anterior (Figura 26), as etapas para instalar e ativar uma montagem de componentes são as seguintes:

- A aplicação de deployment interage com o usuário para determinar onde cada componente será instalado;
- A aplicação de deployment chama o método `install` do objeto `ComponentInstallation`, que obtém o arquivo do componente e opcionalmente o torna disponível no ambiente local;
- A aplicação de deployment cria um objeto `Assembly` chamando o objeto `AssemblyFactory` da máquina onde a montagem deve ser criada, passando como parâmetro a localização do arquivo de descrição da montagem (AAR). Os objetos `Assembly` coordenam a criação e destruição de montagens de componentes;
- Tomando como base a informação contida no arquivo descritor da montagem (AAR), o objeto `Assembly` cria as interfaces *homes* dos componentes, cria cada instancia de componente e faz as conexões entre as instâncias de componentes;
- Para criar um `Component` o objeto `Assembly` usa os objetos `ServerActivator`, `ComponentServer`, `Container`, e `ComponentHome` da forma que será descrita em detalhes nos próximos passos
- O objeto `Assembly` primeiramente chama o objeto `ServerActivator` na máquina alvo para criar o `ComponentServer`. Cada máquina contém uma instância de `ServerActivator`;
- O objeto `Assembly` chama o objeto `ComponentServer` na máquina alvo para criar *containers* dentro do `ComponentServer`. Um *container* é representado pela referência para a interface `Container`;
- O objeto `Assembly` chama o objeto `Container` na máquina alvo para instalar o objeto `ComponentHome` dentro do container;
- O objeto `Container` cria o objeto `ComponentHome`;
- O objeto `Assembly` usa o objeto *ComponentHome* para criar uma instância de *Component*;
- Se aplicável, um *configurator* é aplicado ao objeto `Component` (interface provida pelo usuário que faz a inicialização das propriedades do componente);

- Uma vez que todos os componentes estão instalados, o objeto `Assembly` conecta os componentes que fazem parte da montagem conforme indicado na seção de conexões do arquivo descritor de montagem (AAR); e
- Após a execução com sucesso de cada conexão, o objeto *Assembly* sinaliza para cada objeto envolvido no processo que todas as suas conexões foram bem sucedidas.

## *3.6 Trabalhos Relacionados a Deployment de Componentes*

A literatura tem vários trabalhos a respeito de deployment de componentes. No entanto, trabalhos sobre o Modelo de Componentes CORBA (CCM) e sobre instalação e ativação em CCM ainda são raros.

### 3.6.1. CCM

Wang et al. [19] sugeriram a aplicação de técnicas de reflexão computacional para resolver pontos chaves para a implementação de um CCM com qualidade de serviço (QoS). As técnicas permitem transparência de localização, mudança das propriedades de QoS, mudança do comportamento dos componentes e gerenciamento da utilização de recursos pelos componentes de forma a adaptar o middleware aos requisitos desejados de QoS.

### 3.6.2. Deployment para o CCM

Kebbal e Bernard [16] sugeriram extensões ao processo de deployment do CCM porque notaram duas falhas na especificação CCM. Na opinião deles, essas falhas existem porque o CCM nunca foi implementado e testado efetivamente. Concordamos plenamente com esse ponto de vista.

A primeira falha é "Pobreza" (Poorness): a especificação CCM não lida com alguns assuntos como localização das implementações de componentes e gerenciamento da configuração das máquinas. Eles sugeriram uma ferramenta de busca dos componentes existentes e de monitoramento das máquinas para manter a informação de qual componente já está instalado em qual máquina, bem como a carga de serviço de cada máquina. Isso permitiria ao usuário escolher no momento do deployment qual componente deverá ser instalado em qual

máquina, levando em consideração a reutilização do software que implementa o componente (não precisaria gastar tempo instalando a DLL do componente na máquina, pois ela já estaria lá), bem como a capacidade da máquina executar novas instâncias de componentes. No nosso modelo, nos preocupamos apenas em mostrar para o usuário quais componentes estão disponíveis e deixamos o usuário escolher livremente em qual máquina eles serão instalados. Nosso foco foi definir um modelo que resolvesse a questão do Deployment de Componentes como um todo, enquanto que [16] propõe uma otimização em um dos passos do Deployment que é a escolha do componente e da máquina destino.

Outra falha da especificação CCM apontada por [16] é a "Inexatidão (V*agueness*): ela não descreve uma série de detalhes como a localização dos objetos auxiliares (`Assembly`, `ComponentInstallation` e outros). [16] propõe a utilização de um serviço de Trader ANSA pois ele possui mais recursos. No nosso modelo utilizamos o Serviço de Trader padrão da OMA.

O ORB Mico fez uma implementação do CCM [26] e resolveu a questão do deployment dos componentes desenvolvendo um toolkit específico que só funciona para o mesmo.

O OpenCCM [27] é uma implementação do CCM gratuita. Nessa implementação foram feitas algumas pequenas extensões às classes de deployment do CCM. Além disso, o controle sobre os daemons que controlam o deployment deve ser feito pelo usuário de forma manual (execução e/ou cancelamento de scripts).

### 3.6.3. Deployment de Java Beans

Rudkin e Smith [18] definiram um serviço de criação e um processo de deployment para aplicações em Java que utilizam componentes Java Beans. O processo contém três passos principais: 1) Cadastramento: Os provedores do componente e os clientes do componente se registram no Diretório de Serviços. 2) Definição do Serviço: O provedor do serviço executa todos os passos para construir um serviço baseado em componentes a partir da informação lida do Diretório de Serviços. 3) Instanciação do Serviço: O cliente executa todos os passos para construir uma aplicação utilizando os serviços baseados em componentes disponíveis. Seguindo esses passos, notamos que a lógica do negócio é confiada aos vários participantes do processo.

O deployment proposto por [18] é específico para JavaBeans e não seria possível aplicá-lo para Componentes CORBA. No entanto consideramos que [18] definiu uma seqüência bastante lógica e completa para os passos de um Deployment para JavaBeans. Isso serviu de inspiração para o nosso modelo, onde procuramos definir as etapas do Deployment para Componentes CORBA de forma igualmente clara.

### 3.6.4. Deployment de outros tipos de componentes

Mikic-Rakic e Medvidovic [17] atacaram quatro problemas relacionados a deployment que eles identificaram: 1) deployment inicial de um sistema em uma nova máquina (ou conjunto de máquinas); 2) deployment de uma nova versão do componente em um sistema alvo existente; 3) análise estática, antes do deployment, dos efeitos mais comuns das modificações desejadas no sistema alvo; 4) análise dinâmica, após o deployment, dos efeitos das modificações efetuadas no sistema alvo em execução. Eles utilizaram uma arquitetura de software chamada Prism. Os autores não mencionaram explicitamente que tipo de componentes padronizados (COM, Java Beans ou CCM) Prism suporta. Entretanto, parece que cada um desses tipos de componentes poderia ser suportado se o estilo Prism fosse seguido.

Nosso modelo é específico para Componentes CORBA, e nossas maiores contribuições são em questões de como fazer com que os componentes CORBA sejam instalados, ativados e fiquem prontos para serem utilizados. Nosso foco sempre foi nos componentes CORBA em si, e não nas máquinas (sistemas alvos) onde esses componentes seriam utilizados. Desse modo, consideramos que nosso modelo proposto e o modelo de [17] são bastante distintos, porém são de certo modo complementares, pois enfocam e sugerem soluções para questões diferentes do problema de deployment.

Hissam et al. [15] sugeriram uma nova abordagem, chamada PECT (*Prediction-Enabled Component Technology*), para a tecnologia de componentes que integra a tecnologia de componentes com um modelo de análise. A tecnologia de componentes suporta e enfatiza as premissas utilizadas pelos modelos de análise e provê o meio para deployment das instâncias PECT e dos componentes de software compatíveis com PECT. Nosso modelo proposto é específico para Componentes CORBA enquanto o modelo de [15] é específico para PECT.

Crnkvic, Hnich, Jonsson e Kiziltan [25] concluíram que não há uma base conceitual forte e bem definida para a engenharia de software baseada em componentes e, portanto,

buscaram identificar e formalizar os conceitos relacionados à especificação, implementação e deployment de componentes. [25] aborda o tema componentes de software do ponto de vista teórico somente, sem nenhuma sugestão para solução das questões práticas, enquanto que o nosso modelo proposto busca solucionar a questão do deployment de componentes CORBA.

# Capítulo 4
# Modelo para Instalação e Ativação de Componentes

A especificação de Componentes em CORBA deixa a cargo de cada vendedor a implementação de uma ferramenta de deployment (instalação e ativação) de componentes.

A especificação CCM apresenta uma sugestão de como implementar a ativação de componentes. Como o deployment é deixado em aberto na especificação, essa sugestão de implementação é bastante simplificada, e tem como intuito apenas ilustrar os fatores que precisam ser levados em conta no processo de ativação. No artigo[12], o autor acata a sugestão de implementação da especificação.

Nesse capítulo, descrevemos os requisitos para o processo de deployment e propomos um modelo para deployment de componentes CORBA. Nosso modelo detalha e expande o processo de deployment proposto pelo CCM, discute um Modelo de Informação, possíveis políticas e metodologias para o deployment e o Modelo de um POA para Componentes. Apresentamos ainda um caso de uso ilustrando em detalhes a aplicação do modelo proposto.

Nosso modelo gera possibilidades de implementação mais ricas e abrangentes, principalmente no aproveitamento das diversas políticas do POA conforme a categoria de componente sendo ativada. A definição do Modelo de Informação (criação de um novo repositório) tornou possível preencher as lacunas e os pontos deixados em aberto na especificação, em especial a questão de geração de identificadores e a melhor caracterização do conteúdo das referências para os componentes.

## *4.1 Requisitos do Processo de Deployment*

Os principais requisitos que um Sistema de Deployment especializado para componentes deve atender são:

- Gerenciamento das Referências e Criação dos Componentes: a interface *Home* é o ponto de entrada para o acesso às instancias dos componentes (ver seção 3.5.2.11). Portanto, é

necessário disponibilizar a referência da interface `ComponentHome` para os clientes. No caso dos clientes cientes do uso de componentes, isso é feito através do objeto `HomeFinder`. Para os clientes não cientes do uso de componentes, os serviços de Nomes ou Trader podem ser utilizados. É necessário ainda prover a infra-estrutura básica para a criação e associação de referências para as instâncias de componentes acessadas via a interface Home;

- Gerenciamento de Eventos: a infra-estrutura de deployment deve ser responsável por criar e gerenciar canais de eventos e efetuar as ligações entre as fontes e os sorvedouros dos eventos respectivamente;
- Gerenciamento das Ligações entre os Componentes: a infra-estrutura de deployment deve possibilitar a conexão dos componentes através de suas portas (ligações entre as facetas e receptáculos dos componentes que interagem entre si);
- Gerenciamento do Estado Persistente do Componente: apenas os componentes das categorias Processo e Entidade têm estados persistentes, que podem ser gerenciados pelo container (que interage com o provedor de serviço de persistência) ou pela implementação do componente (o desenvolvedor do componente pode interagir diretamente com o provedor de serviço de persistência através de interfaces disponibilizadas pelo container). A infra-estrutura de deployment cria os containers e ativa os métodos do container responsáveis por salvar e restaurar o estado do componente.

## 4.2 O Modelo Proposto

Nosso modelo de deployment [22, 23] propõe um framework para a instalação e ativação de componentes. As principais diferenças entre nosso modelo e o proposto na especificação CCM são: um diagrama de classes que detalha as partes envolvidas no processo de deployment, um novo repositório chamado *Repositório de Instâncias de Montagens de Componentes (Component Assembly Instance Repository* - CAI), um processo de deployment modificado, a utilização de funcionalidades do POA para a ativação de componentes, e extensões à IOR para componentes.

### 4.2.1. Os Participantes do Modelo

O modelo proposto tem os seguintes tipos de participantes: repositórios, processos e informações do usuário (a participação do usuário está implícita nas peças de informação do usuário). Os repositórios são: Repositório de Interfaces em CIDL, Repositório de Implementação, Assembly Archive (AAR) e Repositório de Instâncias de Montagens de Componentes (*Component Assembly Instance Repository - CAI*). As informações do usuário são: Entrada do usuário e Saída para o Usuário. Os processos são: Ferramenta *Builder*, Populador do Repositório CAI, Gerente de Deployment, Gerente de Instalação, Gerente de Ativação, Servidor de Ativação, Servidor de Eventos, Gerente de Ligações e Servidor de Homes.

### 4.2.2. Repositórios do Modelo Proposto

As seções seguintes explicam os repositórios do modelo proposto que são utilizados pelos processo e que formam as informações do usuário.

#### 4.2.2.1. Repositório de Interfaces em CIDL

É formado pelos *interface repository objects* (os chamados Objetos do IR), organizados hierarquicamente em uma árvore (*containment tree*). O *Component Repository* é um tipo que deriva de *Repository* e é utilizado como container de mais alto nível para os IRs que armazenam informações sobre componentes. A estrutura *Component Repository* está detalhada no Apêndice A.

#### 4.2.2.2. Repositório de Implementação

Contém toda a informação necessária para localizar e ativar implementações de componentes. Já que os componentes são acessados através das suas interfaces equivalentes, o Repositório de Implementação para Componentes é muito similar ao Repositório de Implementação utilizado para localizar e ativar implementações de objetos. A principal diferença é que implementações de objetos são servidores que devem ser executados e implementações de componentes são bibliotecas dinâmicas que devem ser carregadas em memória. Então, ao invés de manter a associação entre um tipo de objeto e um servidor, o Repositório de Implementação guarda a associação entre a definição de um componente e uma

biblioteca dinâmica. Cada vendedor pode especificar livremente o formato do Repositório de Implementações.

#### 4.2.2.3. Assembly Archive (AAR).

O Assembly Archive (AAR) é composto pelos arquivos descritores de componentes (CCD - Corba Component Descriptor), pelo arquivo descritor de montagem de componentes (CAD - Component Assembly Descriptor) e pelos arquivos descritores de propriedades de componentes (CPF - Component Property File). O conteúdo e formato desses arquivos estão definidos na especificação CCM. Resumidamente, o CCD contém a descrição do componente, ou seja, seus atributos, facetas, receptáculos, sorvedouros e fontes de eventos. Já o CAD descreve uma montagem de componentes, quais componentes fazem parte da montagem e como eles estão conectados (ligações facetas X receptáculos, ligações fonte X sorvedouro de eventos). O CPF contém arquivos com as propriedades de cada componente (valores iniciais dos atributos).

#### 4.2.2.4. Repositório de Instâncias de Montagens de Componentes (*Component Assembly Instance Repository* - CAI)

O CAI armazena as informações sobre cada instância de montagem de componentes que passou pelo processo de deployment: a identificação da montagem, características das instâncias de componentes que fazem parte da montagem, e as ligações entre essas instâncias de componentes (via eventos ou via portas). Cada instância de montagem deve ter suas próprias instâncias de componentes. As instâncias de componentes não podem ser compartilhadas pelas instâncias de montagens de componentes.

Cada entrada no repositório CAI corresponde a uma instância de montagem de componentes que foi instalada e ativada. Para cada montagem serão criadas instâncias de seus componentes constituintes. Não existe compartilhamento de uma mesma instância de componente entre duas montagens diferentes.

Seguem os campos do repositório:

- CompAssemblyId: identificação única da instância de montagem de componentes

- CompInstance: registros que descrevem as instâncias de componente que fazem parte da instância da montagem. Para cada instância de componente haverá uma estrutura correspondente com os seguintes campos:
  - CompInstanceId: identificação única da instância do componente
  - CompInstanceTag: string determinada pelo usuário (auxilia na busca pelo componente).
  - DLLName/CreateFunction: nome da DLL que implementa o componente. Nome da função contida na DLL que deve ser invocada para criar uma instância do componente.
  - Hostname/DirectoryDLLFrom: máquina e diretório de onde a DLL que implementa o componente pode ser obtida.
  - Hostname/DirectoryDLLTo: máquina e diretório para onde a DLL que implementa o componente deve ser copiada.
  - Hostname/DirectoryPropertyFileFrom: máquina e diretório de onde o arquivo de propriedades pode ser obtido (campo opcional, condicionado à existência do arquivo de propriedades)
  - Hostname/Directory/PropertyFileTo: máquina e diretório para onde o arquivo de propriedades deverá ser copiado (campo opcional, condicionado à existência do arquivo de propriedades)
  - InterfRepId: Identificação da interface equivalente do componente no repositório de Interfaces
  - ServantRetentionPolicy: política de retenção do servant. Valores possíveis: RETAIN, NO_RETAIN e EXPLICIT.
  - CompCategory: categoria do componente. Valores possíveis: serviço, sessão, processo e entidade.
  - Hostname: máquina onde o componente será instalado e ativado.
  - Referência: referência gerada para a instância do componente.

- EventConnectionInstance: registros que descrevem as conexões entre os componentes através da emissão/recepção de eventos. Para cada conexão haverá uma estrutura correspondente com os seguintes campos:

- ConnectionId: identificação única da conexão.
- ConnectionTag: string determinada pelo usuário (auxilia na busca pelo canal de eventos).
- EventType: nome do tipo de evento sendo passado
- ConnectionSideAId: identificação da instância de componente no lado do emissor do evento, que chamaremos de lado A. Se esse Id for –1 indica que o emissor é uma instância de componente que não faz parte da montagem.
- ConnectionSideARef: referência do componente do lado A da conexão, ou seja, o emissor do evento.
- ConnectionSideZId: identificação da instância de componente do receptor do evento, que chamaremos de lado Z. Se esse Id for –1 indica que o emissor é uma instância de componente que não faz parte da montagem.
- ConnectionSideZRef: referência do componente do lado Z da conexão, ou seja, o receptor do evento.
- EventNameA: nome pelo qual o evento é conhecido pelo componente do lado A (emissor do evento).
- EventNameZ: nome pelo qual o evento é conhecido pelo componente do lado Z (receptor do evento).
- Reference: referência do canal de eventos.

- LinkConnectionInstance: registros que descrevem as conexões entre os componentes através da ligação entre interfaces facetas e interfaces receptáculos. Para cada conexão haverá uma estrutura correspondente com os seguintes campos:
  - ConnectionId: identificação única da conexão.
  - ConnectionSideAId: identificação da instância de componente que contém a faceta, que chamaremos de lado A da conexão. Se esse Id for –1 indica que a faceta pertence a uma instância de componente que não faz parte da montagem.
  - ConnectionSideARef: referência do componente do lado A da conexão (o qual contém a faceta)

- ConnectionSideZId: identificação da instância de componente que contém o receptáculo, que chamaremos de lado Z da conexão. Se esse Id for –1 indica que o emissor é uma instância de componente que não faz parte da montagem.
- ConnectionSideZRef: referência do componente do lado Z da conexão (o qual contém o receptáculo).
- FacetName: nome pelo qual a faceta é conhecida pelo componente no lado A da conexão.
- ReceptacleName: nome pelo qual o receptáculo é conhecido pelo componente no lado Z da conexão.

### 4.2.3. Informação do Usuário no Modelo Proposto

A seguir as partes de informação do usuário, Entrada do Usuário e Saída para o Usuário, são explicadas.

**Entrada do Usuário**

Corresponde a: componentes escolhidos pelo usuário para formar a aplicação, entrada do usuário relativa a montagem, como por exemplo como os componentes escolhidos serão ligados, e entrada do usuário para deployment, ou seja, em quais máquinas cada componente será instalado e ativado.

**Saída para o Usuário**

Corresponde ao `AssemblyId` (nome único da montagem sendo instalada e ativada) e aos `ComponentIds` (identificador de cada instância de componente que faz parte da montagem). O usuário pode usar essa informação para acessar/localizar as instâncias dos componentes que pertencem à montagem como ilustrado na Figura 27.

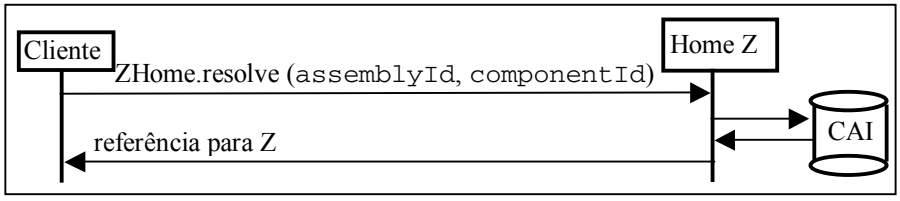


**Figura 27 - Localização dos componentes que fazem parte de uma montagem**

O Cliente solicita à interface Home do componente Z a referência para o componente identificado por `assemblyId,componentId`. A interface Home lê no CAI qual a referência para a instância de componente Z identificada por `assemblyId,componentId` e retorna para o Cliente.

### 4.2.4. Processos do Modelo Proposto

A Figura 28 mostra a dependência entre as classes que implementam os processos do modelo proposto. As seções seguintes explicam os processos: Ferramenta Builder, Populador do CAI, Gerente de Deployment, Gerente de Instalação, Gerente de Ativação, Servidor de Ativação, Servidor de Eventos, Gerente de Ligações e Servidor de Home.

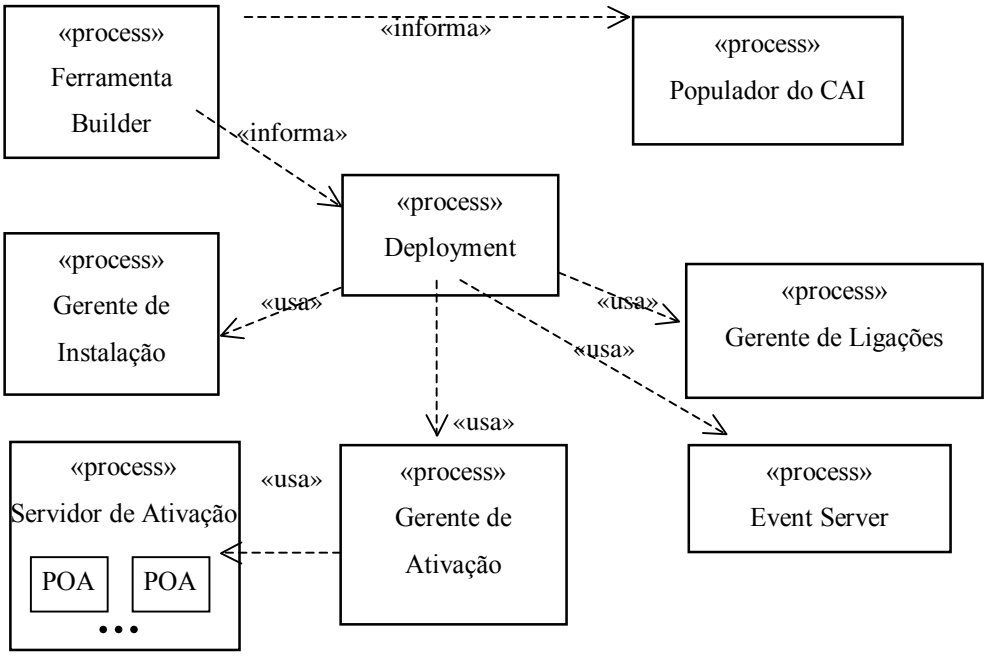


**Figura 28 - Diagrama de classes que mostra a dependência entre os processos da nossa arquitetura de deployment**

#### 4.2.4.1. Ferramenta Builder

Mostra a lista de Componentes baseada nos repositórios de Interface e Implementação e recebe a informação de entrada do usuário. Depois, gera o arquivo Assembly Archive (AAR). A Ferramenta Builder também informa ao Populador do CAI que o arquivo AAR foi gerado. Depois que o Populador do CAI leu o arquivo AAR e atualizou o repositório CAI, a

Ferramenta Builder informa ao Gerente de Deployment que ele pode começar o processo de Deployment. Quando o Gerente de Deployment completa o processo, a Ferramenta Builder retorna para o usuário o `AssemblyId e os ComponentIds`.

Os principais métodos da Ferramenta Builder estão descritos na Tabela VIII.

**Tabela VIII – Métodos da Ferramenta Builder**

| Interface do método | Funcionalidade |
|---|---|
| ⇨ searchForComponents (criteria : String) : ListOfComponents | Mostra a lista de Componentes disponíveis. |
| ⇨ createComponentAssembly() : ComponentAssembly | Inicia o processo de criação de uma instância de montagem de Componentes. |
| ⇨ createCompInstanceIntoCompAssembly (compAss : ComponentAssembly, compDef : ComponentDefinition, compCategory : String, compActivationModel : String) : CompInstance | Cria uma instância de componentes dentro da instância de montagem de componentes. |
| ⇨ setCompInstanceHost (instance : CompInstance, hostname : string) | Indica em qual máquina a instância de componente deve ser instalada e ativada. |
| ⇨ linkCompViaFacetReceptacle (compInstA : CompInstance, compInstB : CompInstance, facetName : string, receptacleName : string) | Indica a conexão de duas instâncias de componentes através de uma ligação faceta e receptáculo. |
| ⇨ linkComViaEvent (comInstSourceId, comInstDestId, eventName) | Indica a conexão de duas instâncias de componentes através de emissão e recepção de um evento. |
| ⇨ generateAARFile (compAssemblyId, filename) | Gera um arquivo AAR contendo todas as informações pertinentes a montagem de componentes: quais os componentes constituintes, como eles estarão conectados e onde serão instalados e ativados. |

#### 4.2.4.2. Populador do CAI

Lê o arquivo AAR gerado e cria uma entrada no repositório CAI representando a montagem sendo instalada e ativada. A principal responsabilidade do Populador é gerar `AssemblyIds` (identificador único da montagem), `ComponentsIds` (identificadores únicos das instâncias de componentes) e `ConnectionIds` (identificadores únicos das conexões entre componentes através de eventos e portas). Todos os identificadores gerados pelo Populador são armazenados no repositório CAI.

Os principais métodos do Populador do CAI estão descritos na Tabela IX.

**Tabela IX – Métodos do Populador do CAI**

| Interface do método | Funcionalidade |
|---|---|
| ⇨ createAssemblyIntoDatabase (AARfileName : String) | Cria entrada no repositório CAI referente a montagem de componentes descrita no arquivo AAR passado como parâmetro. |
| ⇨ deployCompAssembly () : AssemblyIdAndListComponentsIds | Gera os identificadores para a montagem de componentes e para suas instâncias de componentes constituintes. |

#### 4.2.4.3. Gerente de Deployment

Coordena o processo de deployment. É ativado pela ferramenta Builder, recebe como entrada o `AssemblyId`, e o utiliza para indexar a entrada correspondente no repositório CAI e ler as informações necessárias. Interage com os demais gerentes e servidores para criar a infra-estrutura de deployment em cada máquina.

Para cada instância de componente, interage com o Gerente de Ativação para criar a referência CORBA para o componente. Daí, busca na entrada do repositório CAI correspondente por todos os campos que deveriam conter essa referência e os preenche (no registro de CompInstance, o campo Reference; no registro de EventConnectionInstance, os campos ConnectionSideARef ou ConnectionSideZRef (esses campos existem apenas se o componente recebe ou gera eventos); e no registro de LinkConnectionInstance, os campos ConnectionSideARef ou ConnectionSideZRef (esses campos existem apenas se o componente está ligado a outros componentes através de facetas ou receptáculos).

Interage com o Servidor de Eventos para criar canais de eventos para a montagem de componentes e armazena as referências para esses canais no repositório CAI (no registro de EventConnectionInstance, o campo Reference).

Interage com o Gerente de Ligações para "costurar" os componentes que fazem parte de ligações facetas X receptáculos.

O principal método do Gerente de Deployment está descrito na Tabela X.

**Tabela X – Métodos do Gerente de Deployment**

| Interface do método | Funcionalidade |
|---|---|
| ⇨ deploy (AssemblyId) | ⇨ Utiliza o AssemblyId para localizar as informações necessárias no Repositório de Componentes, que serão utilizadas na interação com os demais participantes do processo de deployment.<br>⇨ Interage com o Gerente de Instalação chamando o método:<br>▪ installDLL (hostnameOrig : String, compDLLOrig : String, hostnameDest : String, compDLLDest : String) : boolean<br>⇨ Interage com o Gerente de Ativação chamando o método:<br>▪ createReference (hostname : String, componentCategory : String, componentActivationModel : String, compDLL : String) : ObjRef<br>⇨ Interage com o Servidor de Eventos chamando os métodos:<br>▪ createEventChannel (assemblyId : String, connectionId : String) : ObjRef<br>▪ linkEventProducerConsumer (assemblyId : String, connectionId : String, ObjRef compProducer, String producerName, ObjRef |

| | |
|---|---|
| | compConsumer, String consumerName, String EventType) : void<br>⇨ Interage com o Gerente de Ligações chamando o método:<br>▪ linkFacetReceptacle (assemblyId : String, connectionTag : String, facet : ObjectRef, receptacle : ObjRef) : void |

#### 4.2.4.4. Gerente de Instalação

Distribui as bibliotecas que implementam os componentes e seus arquivos de propriedades (se existirem) para as máquinas onde os componentes serão ativados.

Para cada componente que faz parte da montagem:

- efetua um FTP da biblioteca (DLL) da máquina/diretório origem (campo *Hostname/DirectoryDLLFrom*) para a máquina/diretório destino (campo *Hostname/DirectoryDLLTo*);
- efetua um FTP do arquivo de propriedades (caso exista) da máquina/diretório origem (campo *Hostname/DirectoryPropertyFileFrom*) para a máquina/diretório destino (campo *Hostname/DirectoryPropertyFileTo*).

Os principais métodos do Gerente de Instalação estão descritos naTabela XI.

**Tabela XI – Métodos do Gerente de Instalação**

| Interface do método | Funcionalidade |
|---|---|
| ⇨ installDLL (hostnameOrig : String, compDLLOrig : String, hostnameDest : String, compDLLDest : String) : boolean | Instala biblioteca |
| ⇨ uninstallDLL (hostname : String, compDLL : String) : boolean | Desinstala biblioteca |

#### 4.2.4.5. Gerente de Ativação

Interage com o Servidor de Ativação de cada máquina requisitando a criação de referências para as instâncias de componentes sendo instaladas e ativadas na máquina.

O principal método do Gerente de Ativação está descrito na Tabela XII.

**Tabela XII – Métodos do Gerente de Ativação**

| Interface do método | Funcionalidade |
|---|---|
| ⇨ createReference (hostname : String, componentCategory : String, componentActivationModel : String, compDLL : String) : ObjRef | ⇨ Cria a refêrencia para o componente |

#### 4.2.4.6. Servidor de Ativação

Cria e mantém os POAs que geram as referências para as instâncias de componentes e gerenciam o tempo de vida dos servants que incarnam essas instâncias de componentes.

Para cada categoria de componente e modelo de ativação, um POA específico é criado pelo servidor de Ativação. O modelo de ativação que casa melhor com componentes é o Sob-Demanda: o POA cria a referência para o componente mas o servant só será criado na primeira vez que a referência é utilizada por um cliente. Será utilizado um Gerente de Servant para criar os servants. Esse Gerente de Servant atuará como um container, provendo para o servant o acesso aos serviços CORBA. A política de retenção do servant do POA determinará o comportamento do Gerente de Servant: Localizador de Servant (`NON_RETAIN`) ou Ativador de Servant (`RETAIN`). Outro modelo de ativação útil é o Explícito, no qual no momento de deployment ambos a referência para o componente e o servant são criados. Esse modelo é vantajoso para aplicações que têm poucos componentes e/ou precisam que os componentes estejam disponíveis para uso imediato.

A Tabela XIII sumariza os POAs criados pelo Servidor de Ativação. Para cada par categoria de componente X modelo de ativação é instanciado um POA com políticas específicas e cujo nome é formado pela categoria seguido das letras NR (para Localizador de Servant), R (para Ativador de Servant) ou Exp (para modelo de ativação explícito).

**Tabela XIII - POAs criados pelo Servidor de Ativação**

| Categoria de Componente | Modelo de Ativação | | |
|---|---|---|---|
| | NO_RETAIN | RETAIN | EXPLICIT |
| Serviço | PoaServiceNR | PoaServiceR | PoaServiceExp |
| Sessão | PoaSessionNR | PoaSessionR | PoaSessionExp |
| Processo | PoaProcessNR | PoaProcessR | PoaProcessExp |
| Entidade | PoaEntityNR | PoaEntityR | PoaEntityExp |

A Tabela XIV indica as políticas utilizadas para criar os POAs com o modelo de ativação Sob-Demanda.

**Tabela XIV- Políticas dos POAs com o modelo de ativação Sob-Demanda**

| Políticas / POA | Criação do ObjectId | Ciclo de Vida | Processamento das Requisições | Retenção do Servant |
|---|---|---|---|---|
| ServiceNR | SYSTEM_ID | TRANSIENT | USE_SERVANT_MANAGER | NON_RETAIN |
| ServiceR | SYSTEM_ID | TRANSIENT | USE_SERVANT_MANAGER | RETAIN |
| SessionNR | USER_ID | TRANSIENT | USE_SERVANT_MANAGER | NON_RETAIN |
| SessionR | USER_ID | TRANSIENT | USE_SERVANT_MANAGER | RETAIN |
| ProcessNR | USER_ID | PERSISTENT | USE_SERVANT_MANAGER | NON_RETAIN |
| ProcessR | USER_ID | PERSISTENT | USE_SERVANT_MANAGER | RETAIN |
| EntityNR | USER_ID | PERSISTENT | USE_SERVANT_MANAGER | NON_RETAIN |
| EntityR | USER_ID | PERSISTENT | USE_SERVANT_MANAGER | RETAIN |

No modelo de ativação explícito, a própria aplicação deve criar explicitamente o servant, e solicitar ao POA o cadastro dele no mapa de objetos ativos através dos métodos `activate_object` ou `activate_object_with_id`.

O objetivo é que o servant fique pronto para atender às requisições dos clientes imediatamente. O modelo de ativação explícito é particularmente útil para o caso em que o tempo de resposta deve ser constante, evitando-se o atraso ocasionado pela criação do servant ao chegar a primeira requisição.

Como o servant deve ficar sempre no ar, a política de retenção do servant é sempre RETAIN. Para garantir que nenhum objeto seja automaticamente ativado, ou seja, que o POA buscará apenas por objetos previamente cadastrados no mapa de objetos ativos, a política de Processamento das Requisições é USE_ACTIVE_OBJECT_MAP_ONLY. A Tabela XV mostra as políticas utilizadas para criar os POAs com o modelo de ativação Explícito.

**Tabela XV - Políticas dos POAs com o modelo de ativação Explícito**

| Políticas / POA | Criação do ObjectId | Ciclo de Vida | Processamento das Requisições | Retenção do Servant |
|---|---|---|---|---|
| ServiceExp | SYSTEM_ID | TRANSIENT | USE_ACTIVE_OBJECT_MAP_ONLY | RETAIN |
| SessionExp | USER_ID | TRANSIENT | USE_ACTIVE_OBJECT_MAP_ONLY | RETAIN |
| ProcessExp | USER_ID | PERSISTENT | USE_ACTIVE_OBJECT_MAP_ONLY | RETAIN |
| EntityExp | USER_ID | PERSISTENT | USE_ACTIVE_OBJECT_MAP_ONLY | RETAIN |

Durante o processo de deployment o usuário deve escolher um modelo de ativação (Sob-Demanda Retain, Sob-Demanda Non-Retain ou Explícito) para cada instância de componente. Essa informação é armazenada no repositório CAI.

A referência para um componente é criada pelo POA que corresponde à categoria do componente e ao modelo de ativação desejado, por exemplo: o POA ServiceNR cria as referências para componentes da categoria serviço com ativação NO_RETAIN.

Os POAs não são persistentes. É responsabilidade do Servidor de Ativação criar/inicializar os POAs convenientemente, ou indicar um Adaptador de Ativação (Adapter Activator) capaz de criar os objetos POA posteriormente. Porém, como o Servidor de Ativação é um daemon, que deve estar sempre no ar, e que no momento da sua inicialização recria todos os POAs necessários com os mesmos nomes e políticas, especificar um Adaptador de Ativação tornou-se desnecessário.

O POA cria as referências para os objetos, dentro da qual, além do ObjectId, vai a identificação do POA e da máquina onde o POA foi instanciado. Quando o ORB encaminha uma requisição para um objeto identificado por máquina/POA/ObjectId, e o POA não existe mais naquela máquina, é levantada a exceção OBJECT_NOT_EXIST. A menos que tenha sido

associado ao POA, um Adaptador de Ativação (Adapter Activator), que é um objeto implementado pelo usuário que é capaz de "recriar" o POA desejado. Sempre que o ORB recebe uma requisição para um POA que não existe, ele chama o adaptador de ativação (caso exista), para que o POA seja recriado com o mesmo nome e as mesmas políticas.

Cada POA é identificado por um nome único na máquina que o contém. Os POAs são organizados hierarquicamente, através de relações de parentesco (pai-filho), sendo que o Root POA é o "pai de todos", ou a raiz da árvore, podendo conter vários POAs "filhos".

Deste modo, cada POA representa um espaço de nomes único, que identifica as referências e os servants que o POA gerencia.

Aplicações servidoras portáveis podem assumir que não haverá conflitos de nomes entre os POAs que elas criam e os POAs criados por outras aplicações. Se duas aplicações solicitam a criação de um POA "filho" chamado "Teste" embaixo do RootPOA, a implementação do ORB deve ser capaz de criar ambos os POAs e retorná-los para as aplicações. Isso pode ser feito adicionando-se algum tipo de timeStamp e/ou applicationIdStamp ao nome do POA internamente.

Os principais métodos do Servidor de Ativação estão descritos na Tabela XVI.

**Tabela XVI – Métodos do Servidor de Ativação**

| Interface do método | Funcionalidade |
|---|---|
| ⇨ createReference (componentCategory : String, componentActivationModel : String, compDLL : String) : ObjRef | Cria uma referência. |
| ⇨ destroyReference (ObjRef) : void | Destrói uma referência. |
| ⇨ private init() | Inicializaçao: cria todos os POAs necessários, um para cada par categoria de componente X modelo de ativação. |
| ⇨ private createPOA (POAName : String, componentCategory : String, componentActivationModel : String) : POA | Cria um POA. |

#### 4.2.4.7. Servidor de Eventos

Responsável pela criação do canal de eventos e ligação entre os produtores e consumidores de eventos.

O Gerente de Deployment busca no repositório CAI pelos pares de componentes da montagem que se comunicam através de eventos e decide o número e o nome dos canais de eventos que serão criados. O número varia conforme o algoritmo sendo utilizado, uma possibilidade é criar um canal de eventos para cada par produtor/consumidor (abordagem mais simples), outra possibilidade é minimizar o número de canais sendo criados, reutilizando o canal de eventos para pares de produtores/consumidores coincidentes. O nome do canal de eventos é a concatenação dos campos `AssemblyId` e do `ConnectionId` para garantir unicidade.

O Gerente de Deployment chama o Servidor de Eventos (a máquina do produtor) e solicita a criação do canal de eventos e a ligação dos produtor(es) e consumidor(es). A referência para o canal de eventos criado é armazenada no CAI (campo Reference do registro EventConnectionInstance). Note que os campos ConnectionSideARef e ConnectionSideZRef do registro EventConnectionInstance foram previamente preenchidos pelo Gerente de Deployment durante a geração das referências para as instâncias de componentes. Esse campos são utilizados pelo Servidor de Eventos para fazer a ligação entre o(s) produtor(es) e consumidor(es). Por exemplo, o Servidor de Eventos executa o seguinte pseudocódigo para um evento `Y` emitido pelo componente identificado por compProducerReference que é consumido pelo componente identificado por `compConsumerReference` e o nome do evento no lado do consumidor é `Z`:

```
eventListener = compConsumerReference.get_consumer_Z();
compProducerReference.add_listener_Y(eventListener);
```

O Servidor de Eventos é implementado com um *wrapper* do Serviço de Eventos CORBA e seus principais métodos estão descritos na Tabela XVII.

**Tabela XVII – Métodos do Servidor de Eventos**

| Interface do método | Funcionalidade |
|---|---|
| ⇨ createEventChannel (assemblyId : String, connectionId : String) : ObjRef | Cria canal de eventos |

| | |
|---|---|
| ⇨ linkEventProducerConsumer (assemblyId : String, connectionId : String, ObjRef compProducer, String producerName, ObjRef compConsumer, String consumerName, String EventType) : void | Faz ligação produtor X consumidor do evento |

#### 4.2.4.8. Gerente de Ligações

Componentes podem receber referências para outros objetos e utilizá-las para invocar operações. Os componentes têm como pontos de ligação entre si portas facetas e portas receptáculos. A porta faceta disponibiliza a referência para um objeto. A porta receptáculo é o ponto de ligação com uma referência disponibilizada por uma faceta. O receptáculo é a porta que efetivamente utiliza a referência disponibilizada para invocar métodos.

O Gerente de Deployment descobre no repositório CAI os pares de componentes que devem ser ligados através de portas facetas X receptáculos e solicita ao Gerente de Ligações essa ligação. Note que os campos `ConnectionSideARef` e `ConnectionSideZRef` do registro `LinkConnectionInstance` foram preenchidos previamente pelo Gerente de Deployment durante a criação das referências para as instâncias de componentes. Esses campos são utilizados pelo Gerente de Ligações para ligar os pares faceta X receptáculo executando o seguinte pseudocódigo:

```
facetReference = compFacetProviderReference.provide_Y;
compReceptacleReference.connect_Y (facetReference);
```

O principal método do Gerente de Ligações está descrito na Tabela XVIII.

**Tabela XVIII – Métodos do Gerente de Ligações**

| Interface do método | Funcionalidade |
|---|---|
| ⇨ linkFacetReceptacle (assemblyId : String, connectionTag : String, facet : ObjectRef, receptacle : ObjRef) : void | Efetua a ligação faceta X receptáculo. |

#### 4.2.4.9. Servidor de Home

O usuário pode consultar o Servidor de Home para obter a referência para qualquer componente conforme ilustrado na Figura 27.

O usuário chama o método `resolve(assemblyId,componentId)`, que simplesmente lê no repositório CAI as referências previamente criadas durante o processo de deployment.

O principal método do Servidor de Home está descrito na Tabela XIX.

**Tabela XIX – Métodos do Servidor de Home**

| Interface do método | Funcionalidade |
|---|---|
| ⇨ resolve (assemblyId : String, componentId : String) : ObjRef | Retorna a referência para o componente identificado por assemblyId,componentId. |

## *4.3 Comparação do Modelo Proposto com a arquitetura de Deployment do CCM*

Os participantes do nosso modelo estão relacionados à arquitetura de deployment proposta na especificação do CCM, mostrada na Figura 26, da seguinte forma. As funcionalidades da classe `Assembly` foram distribuídas entre Gerente de Ativação, Gerente de Ligações e Servidor de Eventos. No nosso modelo, o Gerente de Servant do POA atua como a classe `Container` do CCM, provendo para os servants dos componentes acesso aos serviços CORBA. O POA, no nosso modelo, atua de forma similar ao `Component Server` do CCM no sentido que ele chama o Gerente de Servant conforme a política de retenção do servant especificada. Diferente do `ComponentHome` do CCM, o nosso Servidor de Home apenas consulta a referência para o componente. Outra diferença com o `Component` do CCM, é que nosso modelo cria componentes conforme o modelo de ativação.

## *4.4 A Codificação da Referência*

### 4.4.1. Gerenciamento das referências

O padrão CORBA define dois tipos de referências: transientes e persistentes. O tempo de vida de uma referência transiente é limitado pelo tempo de tempo de vida do processo servidor para o qual ela aponta. Uma vez que o processo servidor termina, a referência transiente se torna inválida. Já as referências persistentes existem independentemente do processo servidor, elas identificam uma implementação de objeto cujo ciclo de vida pode perdurar por vários processos servidores. Assim, mesmo que o servidor termine e seja reinicializado, a referência continua válida. É particularmente útil para funcionalidades como ativação automática do servidor ou migração de objeto. Maiores detalhes sobre esse assunto podem ser obtidos em [21].

O POA provê suporte para criação de referências tanto transientes quanto persistentes. A diferença entre elas é que as referências transientes se tornam inválidas quando o POA é desativado. O Servidor de Ativação nunca desativa nenhum POA. Portanto, todas as referências, mesmo as transientes, continuam sempre válidas. É justamente nesse ponto que o conceito de referências transientes e persistentes para componentes se torna um pouco obscuro, pois mesmo as referências transientes parecem "persistentes". A diferença é que quando o POA é criado com a política de Ciclo de Vida com o valor PERSISTENT, deve-se prover um mecanismo que recrie aquele POA sob-demanda (com o mesmo nome e políticas) caso uma referência persistente criada por ele seja utilizada e ele não esteja mais no ar (veja o conceito de Adaptador de Ativação na Seção 4.2.4.6). No nosso modelo isso não é necessário, pois o Servidor de Ativação é um daemon que fica sempre no ar, portanto, os POAs criados por ele estarão sempre disponíveis.

Após criar e ativar cada POA, o Servidor de Ativação fica atendendo solicitações de criação de referências para os componentes (sempre levando em consideração as categorias de componentes e modelo de ativação desejados). Se ele cair, é reinicializado automaticamente, recriando os POAs com os mesmos nomes e políticas.

### 4.4.2. Formato e conteúdo das referências

Referências do padrão CORBA contêm duas seções: identificação de tipo (TypeId) e seqüências de perfis endereçados (Sequence of Tagged Profiles) (Figura 29).

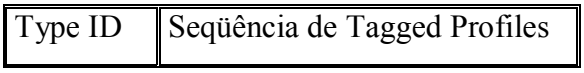


**Figura 29 – Formato Geral da Referência CORBA**

A identificação de tipo (TypeId) é um indicador para o cliente de quais interfaces o objeto referenciado pode suportar. A seqüência de perfis endereçados contém um ou mais perfis que encapsulam informação utilizada pelo protocolo para comunicação com o objeto (máquina, porta, identificador do objeto).

Para o protocolo IIOP1.0, a referência tem o formato apresentado na Figura 30.

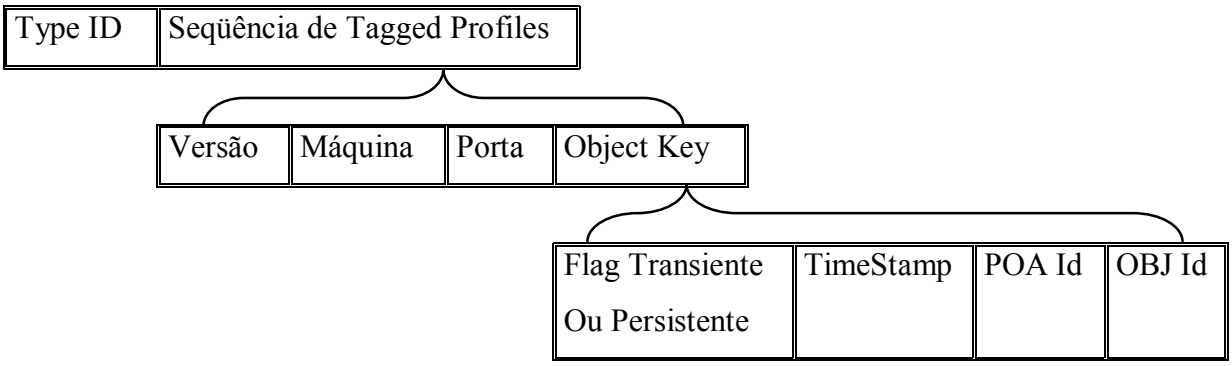


**Figura 30 – Formato da Referência CORBA para IIOP1.0**

Para o protocolo IIOP1.1, foi adicionado o campo Components (que é utilizado pelo IIOP1.1) e dois campos de identificam o fabricante (vendedor) do ORB que gerou a referência e a versão do fabricante do ORB (Figura 31). As referências transientes ainda terão um *TimeStamp* para garantir unicidade, mas as referências persistentes terão um *ServerName* para identificar junto ao Repositório de Implementações a qual objeto/componente elas se referem.

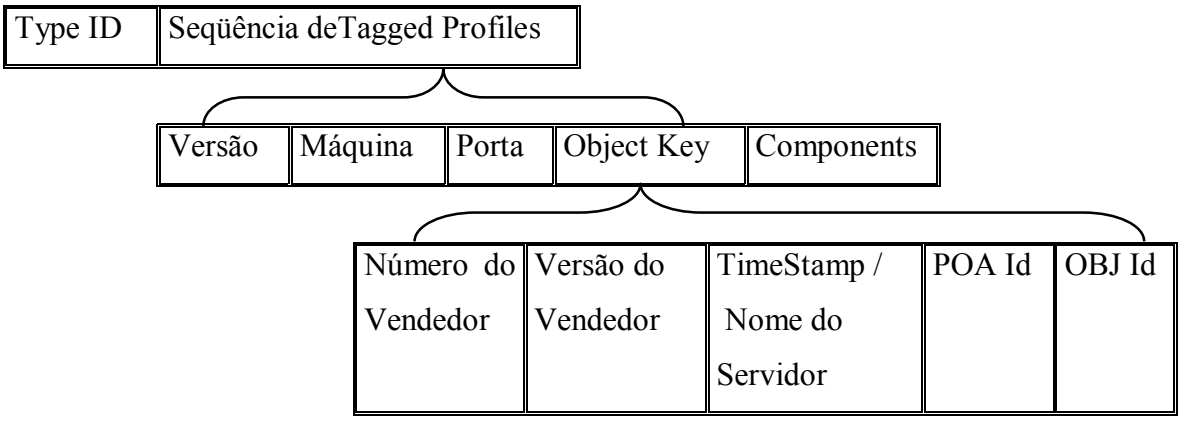


**Figura 31- Formato de Referência CORBA para IIOP1.1**

### 4.4.3. Codificação do ObjectId

Identificamos duas maneiras de associar o nome da DLL e função de criação à referência para o componente.

Na primeira abordagem, toda a referência para um componente conterá no ObjectId uma string com o nome da DLL que implementa o componente e o nome da função que deve ser chamada para que uma instância do componente seja criada (campo *DLLName/CreateFunction* do repositório CAI). Exemplo: `MyServant:create_MyFoo`.

O POA pode ter métodos genéricos para ler essas informações da referência, carregar a DLL em memória e executar a função de criação.

Como o Repositório CAI contém também o nome da DLL e função de criação, uma segunda abordagem possível seria codificar dentro da referência a chave de acesso para o Repositório CAI. Daí o POA poderia ler a chave contida na referência e consultar o Respositório CAI para obter as informações necessárias para criar a instância do componente.

De qualquer modo, um mesmo POA pode ser utilizado para gerenciar/criar/destruir servants de componentes de tipos diferentes, pois a informação a respeito de como criar a instância estará codificada na referência. É justamente esse recurso que torna possível a utilização do Servidor de Ativação com apenas um POA instanciado para cada par categoria de componente X modelo de ativação, portanto, um conjunto mínimo de POAs será criado. Caso contrário, seria necessário um POA para cada tipo de componente, o que tornaria o Deployment inviável.

Note que isso não tem nada a haver com a política do POA de Criação do ObjectId (Id Assignment Policy), cujo valor pode ser USER_ID, ou seja, o ObjectId do servant é definido pela própria aplicação, ou SYSTEM_ID, o ObjectId é definido pelo POA. Em ambos os casos, uma parte do ObjectId conterá o nome da DLL e a função de criação. Faremos uso desse recurso sempre, de modo a tornar genérico o processo de ativação dos Componentes.

## *4.5 Cenários de Ativação*

Os cenários de ativação variam conforme a política de retenção do servant.

### 4.5.1. NO_RETAIN

A Figura 32 ilustra o cenário de ativação NO_RETAIN.

Cada vez que um método é chamado, um novo servant é criado(***1***), a operação relacionada ao método é executada(***2***) e o servant é destruído(***3***).

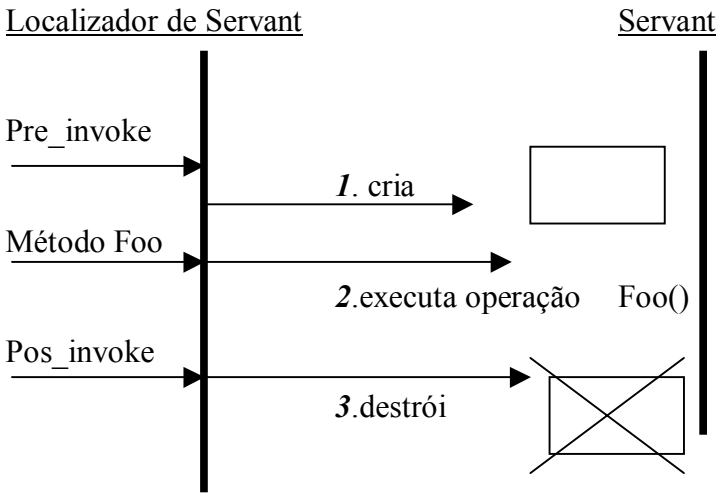


**Figura 32 – Cenário de Ativação NO_RETAIN**

### 4.5.2. RETAIN

A Figura 33 ilustra o cenário de ativação RETAIN.

Na primeira vez que um método é invocado o servant é criado(***1***) e guardado em um mapa de objetos ativos(***2***). Ele só será destruído(***3***) quando o POA que criou sua referência for destruído (POA::Destroy), ou quando o cliente solicitar a desativação do objeto explicitamente (Object::Deactivate).

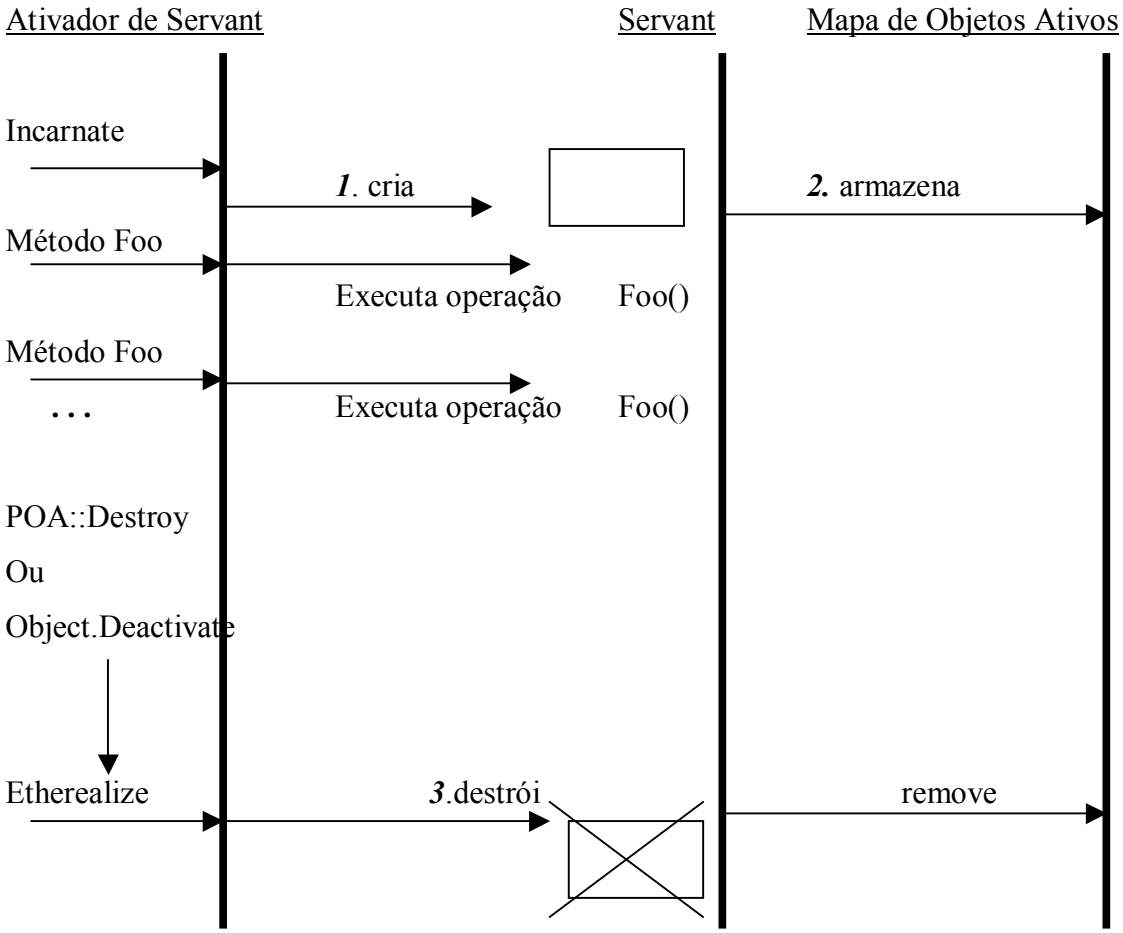


**Figura 33 - Cenário de Ativação RETAIN**

### 4.5.3. Explícito

A Figura 34 ilustra o cenário de ativação explícita.

Os servants são criados explicitamente pela aplicação (***1***,***2***,**3**) e a seguir cadastrados junto ao mapa de objetos ativos do POA (***4***,***5***,***6***). Os servants ficam a postos para executar os métodos. A aplicação decide quando destruí-los (***7***,***8***); por exemplo, através da chamada de POA::Destroy ou Object::Deactivate, como no cenário anterior.

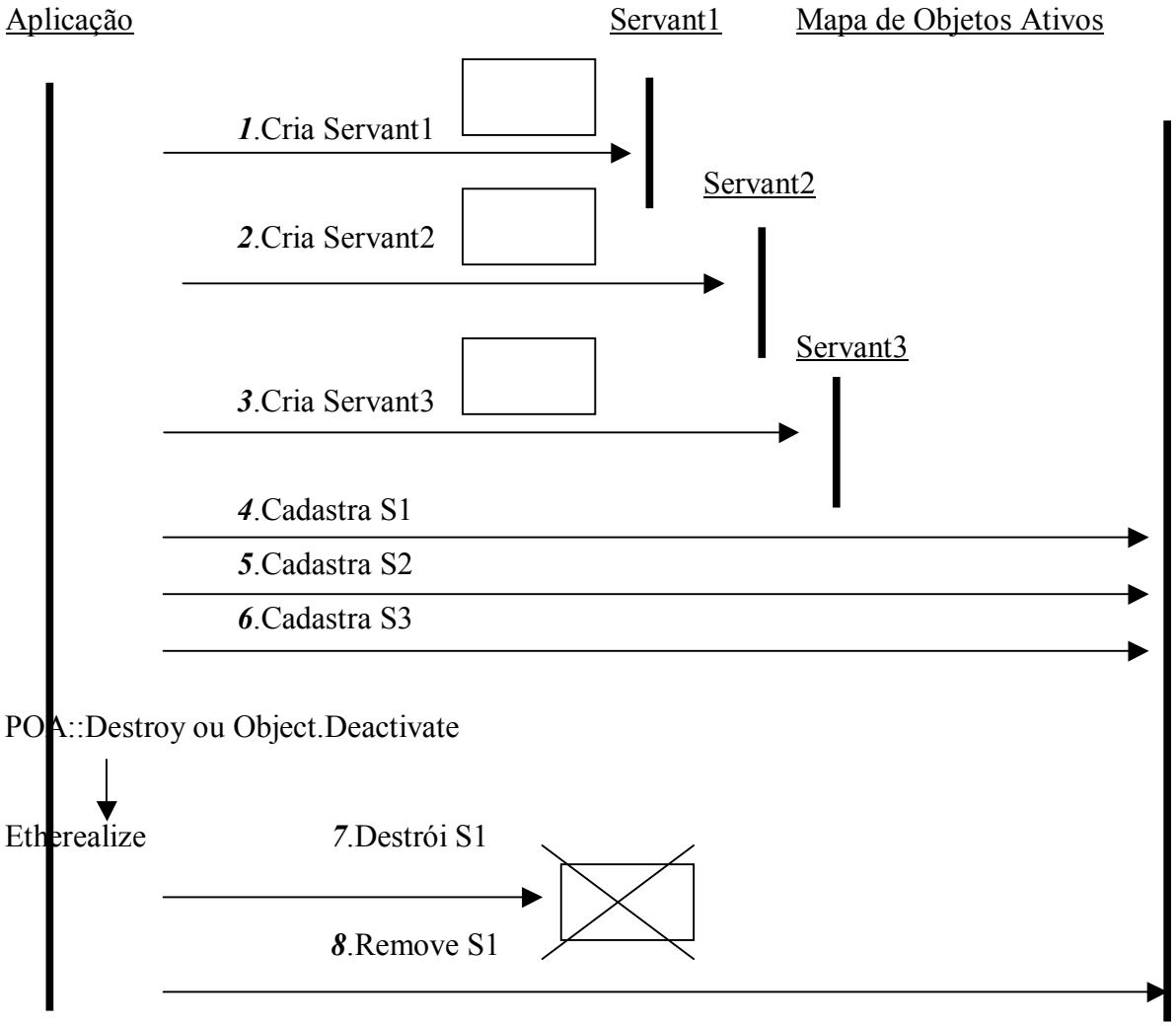


**Figura 34 - Cenário de Ativação Explícita**

## *4.6 Processo de Deployment*

A Figura 35 representa o nosso processo de deployment, que está dividido em fase de montagem (passos *a* até *d*) e fase de deployment (passos *e* até *r*). Esse processo de deployment descreve como juntar a informação necessária para o deployment, e como criar a infra-estrutura de deployment. Nós assumimos que todos os processos de deployment foram disparados e inicializados antes de utilizar a Ferramenta Builder.

### 4.6.1. Fase de Montagem

O usuário ativa a Ferramenta Builder (linha tracejada **1**). A Ferramenta Builder recebe as informações de entrada do usuário: implementações de componentes escolhidas (**a**), nome da montagem que representa a aplicação, a tag de identificação de cada componente (opcional) (**b**) e as máquinas nas quais cada componente deve ser instalado (**c**). Depois, utilizando essas informações a Ferramenta Builder gera o arquivo Assembly Archive (AAR) (**d**).

### 4.6.2. Fase de Deployment

A Ferramenta Builder informa ao Populador do CAI, o nome do arquivo AAR (Assembly Archive) previamente criado (linha tracejada **2**). O Populador do CAI lê o AAR (**e**), cria uma entrada no CAI e gera os `AssemblyId, ComponentIds e ConnectionIds` (**f**). A Ferramenta Builder chama o Gerente de Deployment e informa o `AssemblyId` (linha tracejada **3**). O Gerente de Deployment, usando o `AssemblyId` e as informações contidas no CAI (**g**), interage com: Gerente de Instalação (**h**), que instala as bibliotecas e os arquivos de propriedades (se existirem) nas máquinas alvo; Gerente de Ativação (**i**), que chama o Servidor de Ativação (**j**) em cada máquina para criar as referências para as instâncias de componentes; Servidor de Eventos **(n)**, que cria os canais de eventos e liga os produtores ao consumidores, e Gerente de Ligações **(q)**, que faz a conexão entre os componentes. O usuário pode então utilizar `AssemblyId` e `ComponentIds` armazenados no CAI para acessar os componentes **(r)**. O passo (**j**) simplesmente retorna uma referência de componente para o Gerente de Ativação, que a repassa para o Gerente de Deployment **(l)**, que a armazena no CAI **(m)**. O passo **(o)** retorna a referência de um canal de eventos para o Gerente de Deployment, que a armazena no CAI **(p)**.

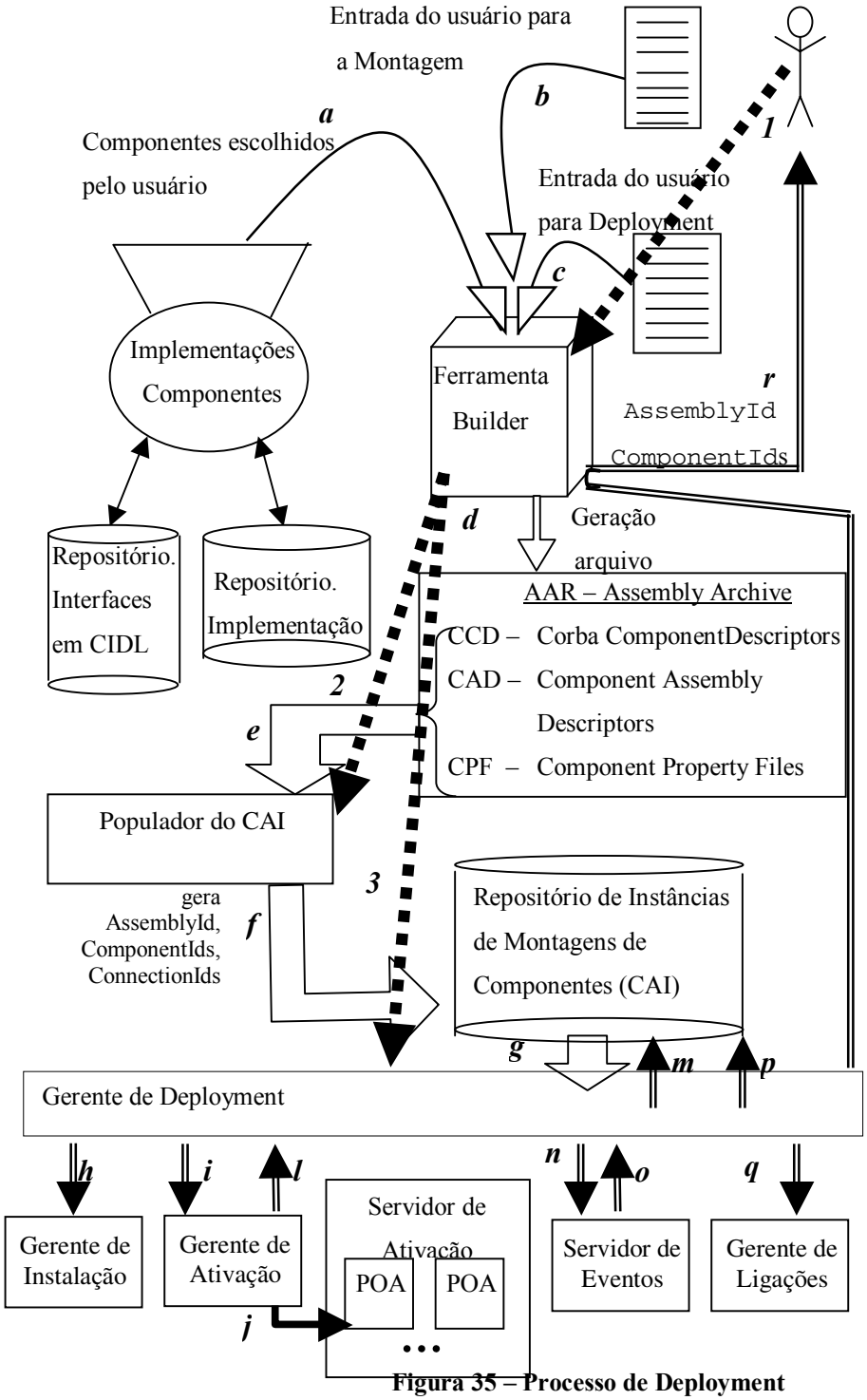


**Figura 35 – Processo de Deployment**

## 4.7 Caso de Uso

Nesta seção nós apresentamos um caso de uso bastante simples de uma montagem de componentes para construir a interface gráfica com o usuário de uma aplicação. A aplicação mostrará duas imagens contendo faces desenhadas (uma à esquerda e outra à direita). Haverá botões para modificar o estado da face à esquerda para triste ou feliz. Daí, através da chamada de método via ligação faceta X receptáculo, o estado anterior da face à esquerda será repassado para a face da direita. Utilizaremos as seguintes declarações de componentes em CIDL:

```
module MyBeans {
     interface Smile {
          void happy();
          void sad()
     }

     component SmileyBean supports Smile {
          void draw();
          provides Smile  actionToBePerformed;
          uses Smile actionToBePerformedAtNeighbour;
          attribute boolean isSmile;
          consumes ButtonPressedEvent actionEvent;
     }

     component ButtonBean {
          void draw();
          attribute string label;
          emits ButtonPressedEvent pressEvent;
     }
};
```

Na Figura 36 verificamos que podemos construir a interface gráfica de uma aplicação juntando quatro instâncias de componentes em uma montagem:

1. *LeftSmile*: o SmileBean que aparece à esquerda

2. *RightSmile*: o SmileBean que aparece à direita
3. *HappyButton*: o ButtonBean que tem o dizer “Happy”
4. *SadButton*: o ButtonBean que tem o dizer “Sad”,

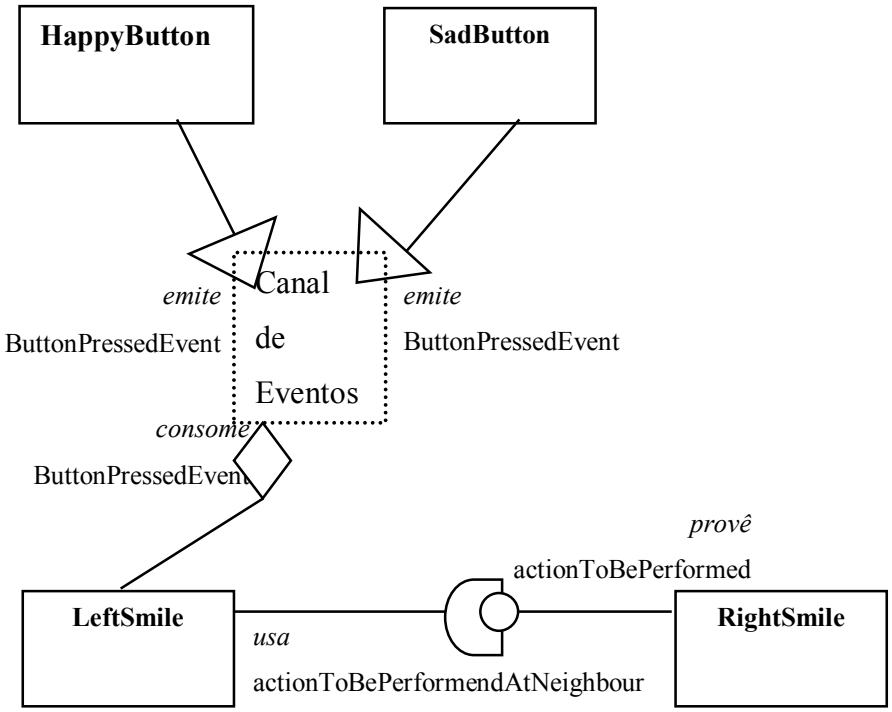


**Figura 36 – Montagem de componentes para formar a interface gráfica de uma aplicação**

Graficamente, nossa interface com o usuário seria semelhante à Figura 37.

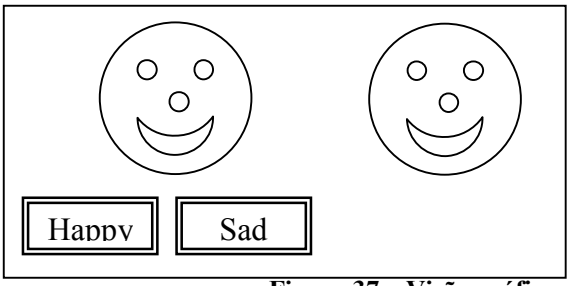


**Figura 37 – Visão gráfica da interface com o usuário**

Nossa aplicação terá o seguinte comportamento:

- Arquivos de propriedades serão utilizados para inicializar o estado inicial das faces *LeftSmile* e *RightSmile*
- Os botões *HappyButton* e *SadButton* modificam o estado da face *LeftSmile*.
- Quando o estado da face *LeftSmile* é modificado, ele repassa seu estado anterior para a face *RightSmile*.

Por exemplo, se o *LeftSmile* está "triste", e o *RightSmile* está "feliz" (Figura 38) e o botão *HappyButton* é pressionado, um evento *ButtonPressedEvent* é enviado à face *LeftSmile*. Daí *LeftSmile* muda seu estado para "feliz" e envia seu estado anterior ("triste") para a face *RightSmile* (Figura 39).

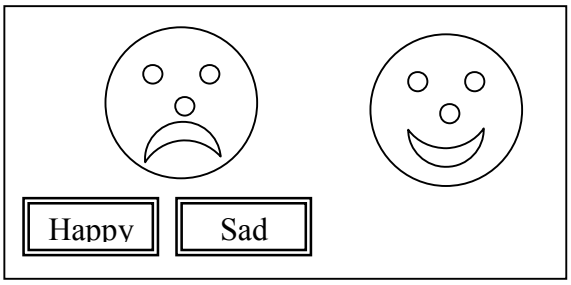


**Figura 38 – Comportamento da Interface Gráfica – estado inicial**

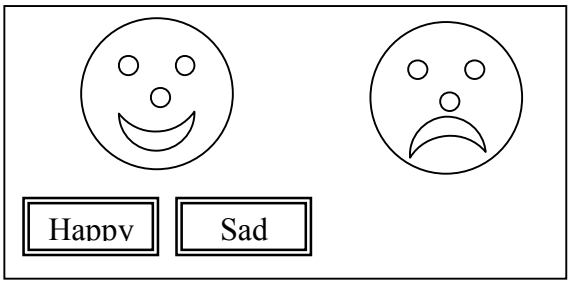


**Figura 39 – Comportamento da Interface Gráfica – após pressionar o botão *HappyButton***

Vamos seguir os passos do processo de deployment conforme ilustrado na Figura 35 para esse caso de uso:

**(a)** Componentes escolhidos pelo usuário: O usuário escolhe as implementações de componentes a serem utilizadas: `SmileyBean e ButtonBean` , e informa quais instâncias devem ser criadas: duas instâncias de `SmileyBean` (LeftSmile e RightSmile) e duas

instâncias de `ButtonBean` (HappyButton e SadButton), e qual o estado inicial de cada instância (arquivos de propriedades). Cada implementação de componente detêm a informação da localização da biblioteca e função de criação para criar uma instância daquele componente (hostname, directory\DDL_name\create_function);

**(b)** Entrada do usuário para Montagem: O usuário liga as instâncias de componentes: os eventos emitidos por HappyButton e SadButton são enviados para LeftSmile, há uma relação faceta X receptáculo entre LeftSmile e RightSmile;

**(c)** Entrada do usuário para Deployment: O usuário informa em qual máquina cada instância de componente deve ser instalada e qual a categoria e modelo de ativação de cada instância de componente. Para o nosso caso de uso todas as instâncias serão da categoria serviço, terão o modelo de ativação NO_RETAIN e serão instaladas na máquina gaivota;

**(d)** Geração do arquivo AAR: O arquivo AAR é gerado no formato definido na especificação, contendo toda a informação reunida nos passos anteriores;

**(e,f)** Populador do CAI: lê o arquivo AAR e cria uma entrada no repositório CAI descrevendo a montagem. O populador é responsável por gerar identificadores únicos para os campos `AssemblyId, ComponentsIds` e `ConnectionIds.` A entrada do repositório CAI para o caso de uso está ilustrada no Apêndice B;

**(g,h)** Gerente de Deployment: lê o conteúdo da entrada do Repositório CAI e para cada instância de componente chama o Gerente de Instalação passando as informações dos campos Hostname/DirectoryDLLFrom e Hostname/DirectoryDLLTo. Note que o Hostname nesses campos é igual ao campo Hostname que é a máquina onde a instância de componente deve ser instalada e ativada;

**(i)** Para cada instância de componente, o Gerente de Deployment chama o Gerente de Ativação usando a informação dos campos: Hostname, CompCategory, Activation Model/ ServantRetentionPolicy e DLLName/CreateFunction. O Gerente de Ativação chama o Servidor de Ativação **(j)** da máquina especificada pelo Hostname solicitando a criação da referência. Com a referência retornada **(l)**, o Gerente de Deployment busca na entrada do repositório CAI por todos os campos que deveriam conter a referência gerada e os atualiza **(m)**: no registro CompInstance atualiza o campo Reference; se a instância de componente é parte de uma conexão via eventos nos registros EventConnectionInstance atualiza os campos ConnectionSideARef ou ConnectionSideZRef; e se a instância é parte de uma conexão via

receptáculoXfaceta nos registros LinkConnectionInstance, os campos ConnectionSideARef ou ConnectionSideZRef são atualizados;

**(n)** Para conexões via eventos, o Gerente de Deployment chama o Servidor de Eventos para criar canais de eventos e armazena as referências retornadas **(o)** na entrada do repositório CAI;**(p)** O Gerente de Deployment preenche nos registros EventConnectionInstance o campo Reference;

**(q)** O Gerente de Deployment chama o Gerente de Ligações para ligar os componentes que estão conectados via portas facetas X receptáculos; e

**(r)** Finalmente, o `AssemblyId` e os `ComponentIds` são retornados para o usuário, que pode utilizar essas informações para acessar/localizar as instâncias de componentes que fazem parte da montagem, conforme ilustrado na Figura 27.

# Capítulo 5
# Implementação e Testes

Foram implementadas versões simplificadas do Populador do CAI, do Servidor de Eventos, do Gerente de Ligações, e uma versão com todas as funcionalidades do Servidor de Ativação, com o objetivo de validar o modelo e avaliar alguns aspectos de implementação e performance relacionados à plataforma distribuída CORBA.

## *5.1 Ambiente de Desenvolvimento*

O nosso ambiente de desenvolvimento foi composto de estações de trabalho SUN com sistema operacional Solaris 2.7 do Instituto de Computação.

As estações ficaram um tempo alocadas exclusivamente para esse trabalho, a fim de garantir a acuidade das medidas e minimizar os fatores de interferência (concorrência de recursos computacionais com outros softwares e/ou usuários).

Usamos o TAO, que é um ORB de domínio público compatível com a especificação CORBA 2.X desenvolvido pelo grupo de computação distribuída (Distributed Object Computing) da Universidade de Washington, baseado na arquitetura de tempo real ACE. O TAO está disponível para download em [13].

O TAO utiliza o protocolo IIOP1.0 para comunicação entre ORBs, e provê uma implementação do POA (Portable Object Adapter), além de conter algumas extensões para suporte a tempo real.

Escolhemos o TAO para a nossa implementação porque ele suporta o POA, que é a peça central do nosso modelo de deployment de componentes.

TAO foi portado para uma variedade de sistemas operacionais, entre eles: Windows NT, várias versões de UNIX (Solaris, Linux, SCO, entre outras) e sistemas de tempo real (VxWorks, Chorus).

## 5.2 Aspectos de implementação

Todos os participantes do modelo (Gerente de Deployment, Gerente de Instalação, Gerente de Ativação, Servidor de Eventos, Gerente de Ligações) são objetos CORBA implementados como *daemons* porque eles precisam estar sempre disponíveis. As interfaces de cada participante estão descritas na Seção 4.2.4.

Como vimos anteriormente, há uma instância de POA para cada par categoria de componente X modelo de ativação, porque cada par requer um conjunto específico de políticas de POA.

No modelo de ativação Sob-Demanda, a aplicação pode registrar um Gerente de Servant, que será invocado pelo POA para obter um Servant. O Gerente de Servant é capaz de encarnar ou ativar um servant e retorná-lo para o POA. Existem dois tipos de Gerente de Servant: Ativador de Servant (no caso do POA ter a política de RETAIN), e Localizador de Servant (para a política NO_RETAIN). No nosso modelo, as implementações de Localizador de Servant e Ativador de Servant são genéricas, ou seja, elas podem lidar com qualquer tipo de componente, porque a informação específica de cada tipo de componente (nome da DLL e função de criação) está encapsulada dentro da referência (implementa a primeira proposta da seção 4.4.3).

A classe que implementa o Gerente de Servant serve como classe auxiliar para as classes Localizador de Servant e Ativador de Servant, contendo os seguintes métodos em comum utilizados por ambas as classes:

- **`create_dll_object_id (char* libname, char* factory_function) : ObjectId`** – cria um ObjectId contendo o nome da DLL e da função de criação;
- **`obtain_servant (char* str, POA poa) : Servant`** – obtém um servant para ativação, carregando a DLL apropriada em memória e chamando a função de criação do objeto servant. O parâmtero <str> é um ObjectId que contém o nome da DLL e da função de criação;
- **`destroy_servant (Servant servant, ObjectId &oid) : void`** – o servant é destruído e a DLL que estava ligada dinamicamente é fechada.

A classe Localizador de Servant é invocada quando o POA tem as políticas USE_SERVANT_MANAGER e NO_RETAIN, e tem os seguintes métodos principais:

- **`preinvoke (ObjectId &oid, POA poa, char* operation, Cookie cookie) : Servant`** – Esse método é invocado pelo POA sempre que é recebida uma requisição para um objeto não-ativo. Note que o parâmetro <operation> indica a operação a ser invocada no Servant.
- **`posinvoke (ObjectId &oid, POA poa, const char* poeration, Cookie cookie) : void`** – Sempre que o Servant completa uma requisição, esse método é invocado para destruir o Servant.

A classe Ativador de Servant é invocada quando o POA tem as políticas USE_SERVANT_MANAGER e RETAIN, e tem os seguintes métodos principais:

- **`incarnate (ObjectId &oid, POA poa) : Servant`** – Quando é recebida uma requisição, o POA primeiramente consulta o mapa de objetos ativos, caso o Servant correspondente ao ObjectId não seja encontrado no mapa, o POA invoca o método **`incarnate`** para obtenção de um Servant (solicitação à classe Gerente de Servant) e armazenamento do Servant no mapa de objetos ativos. Daí a requisição é encaminha ao Servant.
- **`etheralize (ObjectId &oid, POA poa,...) : void`** – Esse método é invocado sempre que o objeto é desativado ou quando o POA é destruído.

No apêndice C estão as declarações completas dessas classes Gerente de Servant, Ativador de Servant e Localizador de Servant.

Outra questão era como nomear os canais de eventos criados. Para garantir unicidade, cada canal de eventos foi nomeado com a concatenação de `AssemblyId` e `ConnectionId`. Esses identificadores são únicos e gerados em ordem crescente pelo Populador do CAI.

É fundamental ressaltar que o Gerente de Deployment desempenha o papel importante de propagar as referências de componentes geradas para todos os campos pertinentes da entrada do repositório CAI.

## 5.3 Testes

Essa seção mostra alguns resultados de testes feitos com todos os POAs definidos na Tabela XIII - POAs criados pelo Servidor de Ativação.

### 5.3.1. Componente Básico utilizado para os testes

Os testes de ativação visam um estudo comparativo entre os diversos POAs definidos no nosso modelo. Utilizamos um cenário padrão, onde a implementação de um componente é encapsulada em uma DLL, e observamos como cada POA gerencia o processamento das requisições e a criação/binding/deleção dos servants para o componente.

Definimos um componente muito simples com apenas uma operação `Long doit ()` que retorna uma constante do tipo inteiro. Com isso garantimos que o componente não consome praticamente nenhum recurso da máquina (memória, cpu) e conseguimos isolar melhor o impacto das atividades dos POAs na performance geral.

Foram feitas medidas do tempo gasto para processar a requisição para cada POA. Para obter o tempo gasto (em milisegundos) entre a invocação de uma operação e o seu retorno, utilizamos uma classe do ACE+TAO chamada ACE_Profile_Timer.

### 5.3.2. Cenário de Testes

Para cada POA, nós desenvolvemos um experimento com três execuções, contendo respectivamente 10, 100 e 1000 invocações à operação `Long doit()`.

Em cada execução, nós medimos a média de tempo de todas as invocações, o tempo da primeira invocação e a média de tempo das demais invocações. Para obter resultados mais acurados, fizemos a média entre três rodadas de cada execução. Ainda, para isolar melhor cada cenário e garantir que não havia interferências entre as execuções, para cada execução um novo cliente era criado no início da execução e destruído no final, e um novo Servidor de Ativação contendo apenas um dos POAs específicos era criado antes da primeira invocação e destruído ao final da última invocação. Os resultados estão ilustrados nas Figuras de Figura 40 a Figura 44. Adicionalmente, os experimentos foram executados colocando o cliente em uma estação de trabalho diferente do servidor, mas os resultados obtidos foram semelhantes aos com cliente e Servidor de Ativação na mesma máquina.

Todos os valores a seguir que utilizaremos nos comentários são médias (das três execuções).

Na Figura 40 estão ilustrados o tempo gasto na primeira invocação e a média de tempo nas demais invocações, para um total de 10 invocações.

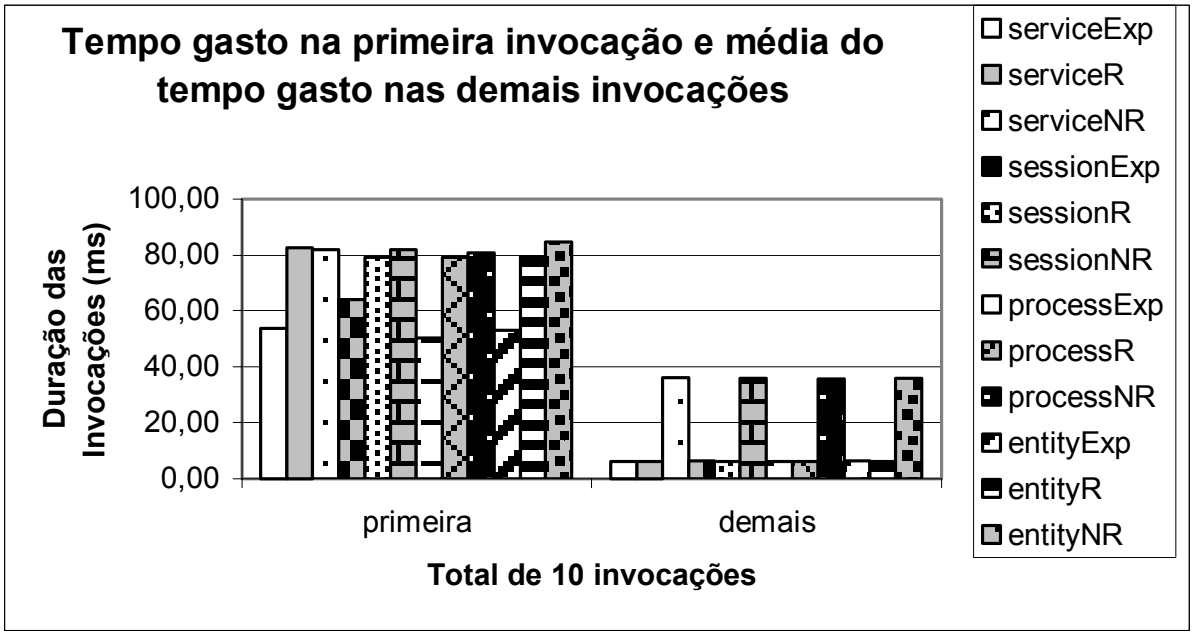


**Figura 40 – Tempo gasto na primeira invoação e média do tempo gasto nas demais invocações (total de 10 invocações)**

A Figura 41 ilustra o tempo gasto na primeira invocação e a média de tempo nas demais invocações, para um total de 100 invocações.

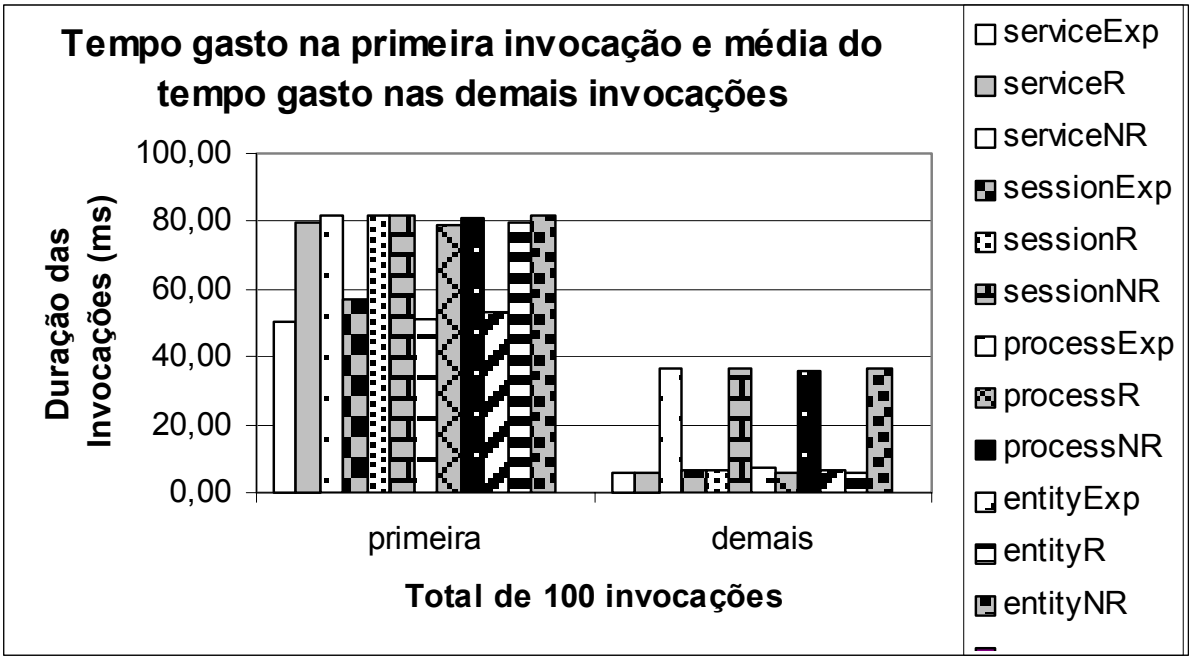


**Figura 41 – Tempo gasto na primeira invoação e média do tempo gasto nas demais invocações (total de 100 invocações)**

A Figura 42 mostra a mesma informação contida na Figura 41, mas coloca lado a lado o tempo gasto na primeira invocação e a média de tempo nas demais invocações, para um total de 100 invocações

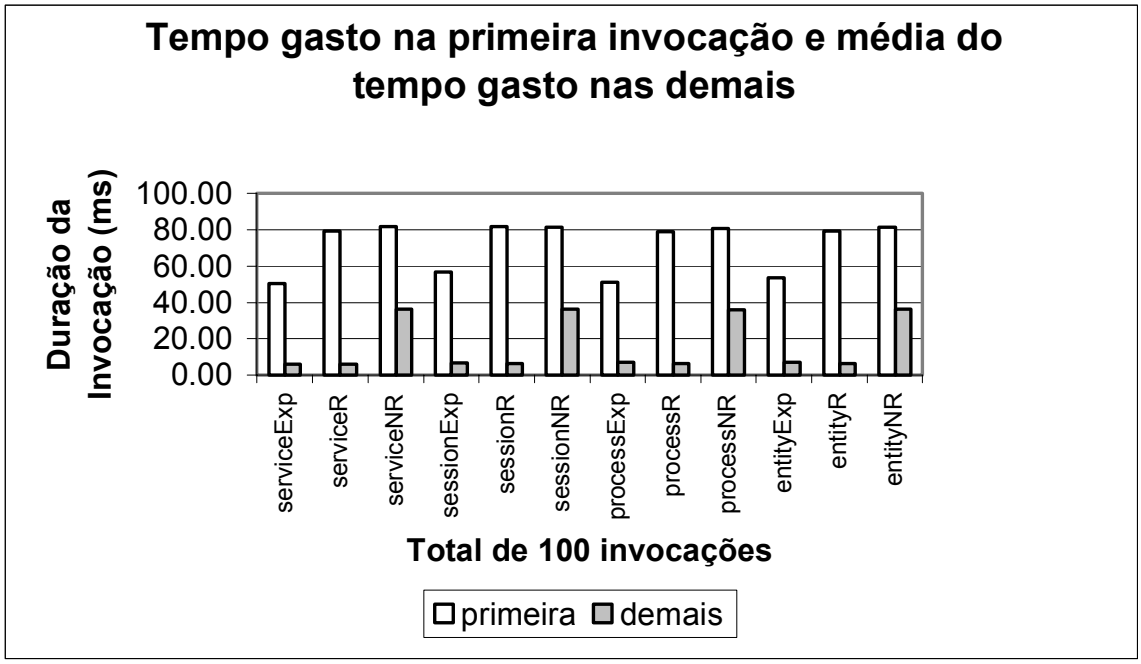


**Figura 42 - Média de tempo da primeira invocação e das demais para execuções com 100 invocações**

A Figura 43 ilustra o tempo gasto na primeira invocação e a média de tempo nas demais invocações, para um total de 1000 invocações.

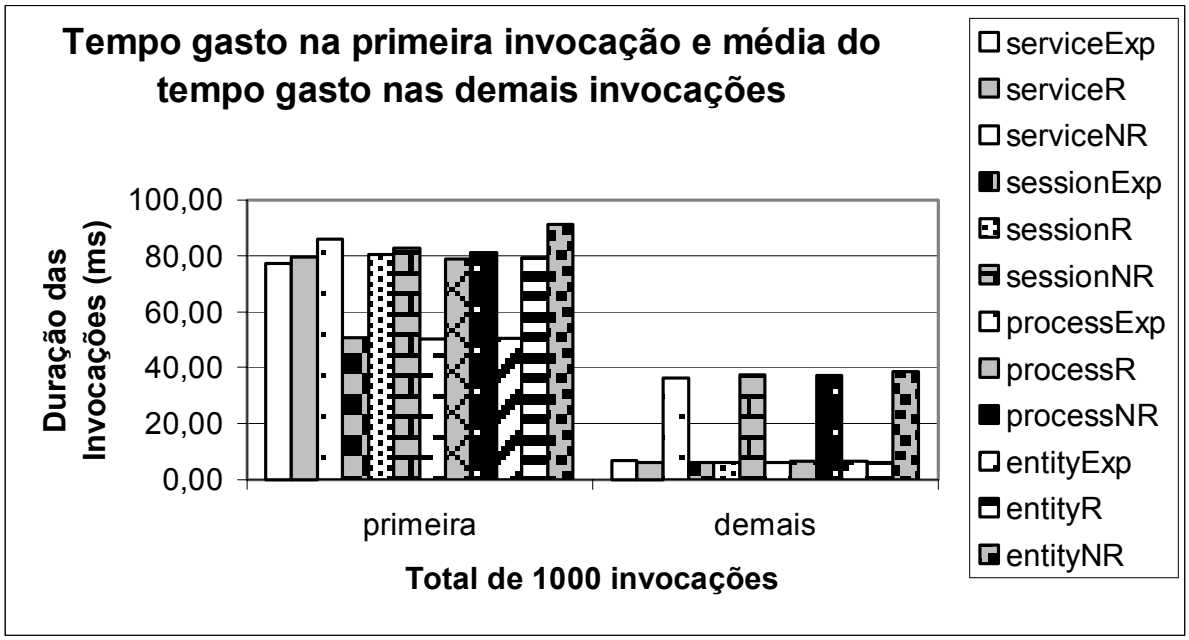


**Figura 43 – Tempo gasto na primeira invoação e média do tempo gasto nas demais invocações (total de 1000 invocações)**

A Figura 44 mostra um panorama geral, ilustra o tempo gasto de todas as invocações, para um total de 10, 100 e 1000 invocações.

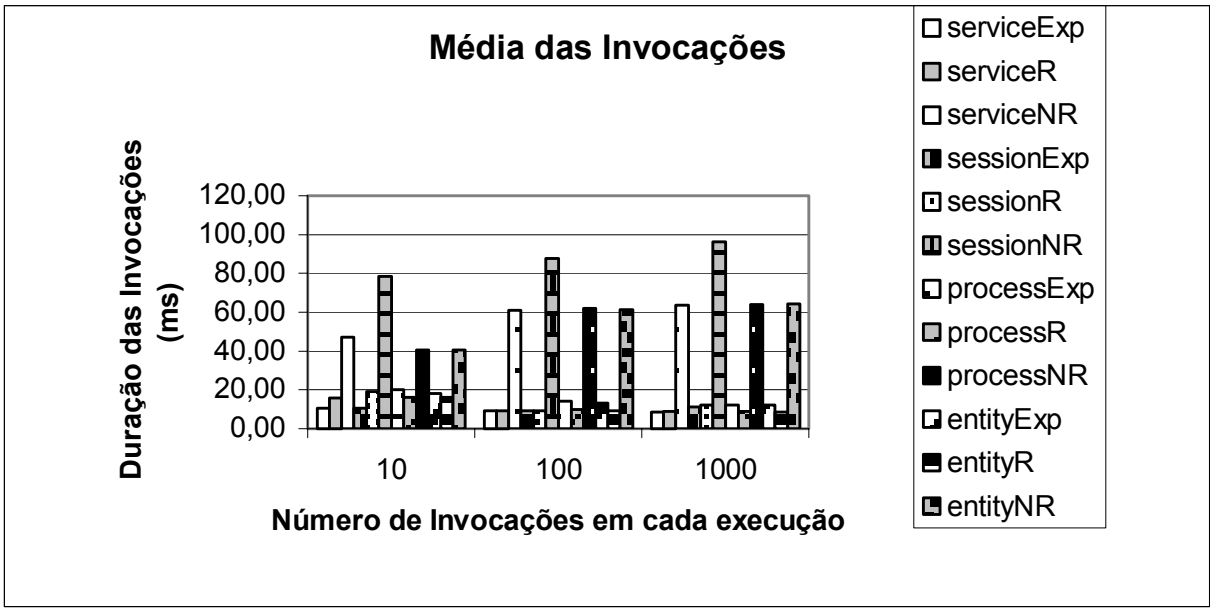


**Figura 44 - Média de tempo de todas as invocações para as execuções com 10, 100 e 1000 invocações**

Nela podemos verificar que os dados obtidos são bastante semelhantes para as três quantidades de invocações, pois o que de fato rege a variação dos valores é o tipo do POA que atende às invocações.

Portanto, para analisar os valores obtidos, comparando as semelhanças ou diferenças entre os tipos de POAs, podemos escolher qualquer um dos gráficos anteriores. Escolheremos a Figura 41 que mostra os resultados da execução com 100 invocações.

A Figura 41 mostra claramente que os POAs com política RETAIN (POAServiceR, POASessionR, POAProcessR, POAEntityR, POAServiceExp, POASessionExp, POAProcessExp, e POAEntityExp) apresentam tempos de invocação (das demais invocações) mais baixos (6.4 ms) do que os POAs com políticas NO_RETAIN (36.15 ms). Isso confirma o que intuitivamente já se esperava, pois com a política RETAIN o servant é criado uma única vez na primeira invocação e reutilizado nas demais invocações. Também está claro que na primeira invocação da execução, os POAs com modelo de ativação explícita (53 ms) são melhores do que os com modelo de ativação sob-demanda com retenção do servant RETAIN

(79.80 ms) porque no modelo de ativação, o processamento da requisição é USE_ACTIVE_OBJECT_MAP_ONLY e no sob-demanda é USE_SERVANT_MANAGER. Em outras palavras, no modelo de ativação explícita apenas o Mapa de Objetos Ativos é consultado, enquanto que no sob-demanda o Mapa de Objetos Ativos é consultado e daí um Gerente de Servant é chamado para a obtenção/criação do servant. Outra característica é que o tempo da primeira invocação é muito próximo para os POAs com ativação sob-demanda e políticas de retenção do servant RETAIN (79.80 ms) e NON_RETAIN (81.50 ms). Isso ocorre porque ambas as políticas têm que inicializar o POA na primeira invocação a fim de se preparar para as próximas invocações. A política NON_RETAIN apresenta um overhead um pouco maior do que a RETAIN, na primeira invocação, porque é preciso destruir o objeto após cada invocação.

Particularmente, técnicas de otimização [8] que foram aplicadas à implementação do POA na TAO melhoram, por exemplo, o gerenciamento de memória (uso de design pattern de armazenamento por thread) e a demultiplexação do POA (hashing perfeito e demultiplexação ativa). Essas técnicas de otimização melhoraram a diferença entre a primeira e as demais invocações nos POAs com políticas de retenção do servant RETAIN e NON_RETAIN. Além disso, a enorme diferença entre a primeira e as demais invocações para a política RETAIN ocorre devido à característica instrínsica da política RETAIN, mas também porque no TAO, a política RETAIN se beneficia da aplicação de um mapa de reverse-lookup adicional que associa cada servant ao seu ObjectId em ordem de $O(1)$ [8]. Em geral, a primeira invocação é aproximadamente ((79.80+81.50)/2)/6.40 = 12.6 vezes mais lenta do que as demais nos POAs com política de retenção do servant RETAIN, e é aproximadamente 81.50/36.15 = 2.25 vezes mais lenta nos POAs com política de retenção do servant NON_RETAIN.

Finalmente, de acordo com os experimentos de performance efetuados, podemos concluir que a melhor performance corresponde aos POAs com modelo de ativação explícito, que são 79.8/53 = 1.5 vezes mais rápidos na primeira invocação e são similares nas invocações seguintes, se comparados com os POAs com modelo de ativação sob demanda e política de retenção do servant RETAIN. Entretanto, dependendo da aplicação, não é razoável criar todos os componentes a priori (na inicialização da aplicação), principalmente por causa do alto consumo de recursos computacionais. Então, uma aboradagem mais razoável seria utilizar o

modelo de ativação explícito apenas para os componentes que são muito utilizados e que demandam um tempo de resposta muito rápido.

A pior performance foi obtida com os POAs com modelo de ativação sob demanda e retenção do servant NO_RETAIN, que são 79.8/53 = 1.5 vezes mais lentos na primeira invocação e 36.15/6.40 = 5.65 vezes mais lentos nas invocações subseqüentes, se comparados com os POAs com ativação sob demanda e política de retenção do servant `RETAIN`. Esse modelo de ativação deveria ser escolhido para os componentes que não são usados com muita freqüência.

# Capítulo 6
# Conclusões

A utilização de componentes de software possibilita acelerar o tempo de desenvolvimento e aumentar a qualidade, pois o software não é mais desenvolvido do zero, mas sim a partir da montagem de componentes cuja implementação já foi bem testada e está robusta e confiável. Além disso, o desenvolvedor se ocupa somente com as questões relacionadas à lógica da aplicação sendo desenvolvida, já que o ORB cuida da parte de infra-estrutura e comunicação entre os componentes de forma transparente.

Como o deployment de componentes CORBA é um assunto importante para o uso de componentes, nessa dissertação propomos um modelo de deployment estendido que detalha os participantes envolvidos no processo de deployment (repositórios, processos e informação do usuário), um processo de deployment modificado que detalha: todas as interações entre os participantes do modelo e entre o usuário e o modelo, um novo repositório que auxilia o deployment armazenando informações a respeito de conexões (através de portas e eventos) entre os componentes, e uma extensão à IOR que possibilita a utilização de Gerentes de Servant genéricos. Essa dissertação também apresenta um caso de uso que detalha o uso do modelo proposto, e alguns aspectos da implementação do modelo proposto usando o ORB TAO.

Foi feito o mapeamento entre as categorias de componentes e os POAs mais indicados para atender às necessidades de cada categoria, incluindo a definição das políticas a serem utilizadas na criação de cada POA. Algumas partes do modelo de deployment proposto foram implementadas com o objetivo de avaliar se esse mapeamento apresentava o comportamento desejado e qual era o impacto na performance conforme o modelo de ativação escolhido para o POA. Com isso, conseguiu-se verificar que o modelo de ativação explícito é o que apresenta a melhor performance e, portanto, é o mais indicado para os componentes que demandam um tempo de resposta muito rápido. Já a pior performance foi obtida com os POAs com modelo de ativação sob demanda e retenção do servant NO_RETAIN, o que indica que esse modelo de ativação deve ser escolhido para os componentes que não são usados com muita freqüência.

Em caso de falha, o modelo proposto auxilia na recuperação: quando o Servidor de Eventos é inicializado ele recria todos os POAs, e quando o Servidor de Eventos é inicializado ele lê do Repositório CAI as informações necessárias para recriação e reconexão dos canais de eventos.

Além disso, o Repositório CAI contém toda a informação para desfazer o deployment, sendo que apenas os processos nas máquinas necessárias são notificados e participam do cancelamento do deployment.

Uma desvantagem do nosso modelo é a dependência do Repositório CAI. Se novas categorias de componentes com novas políticas de gerenciamento de ciclo de vida do Servant são adicionadas ao sistema, nosso modelo precisa ser atualizado. Nesse sentido, a arquitetura de deployment do CCM é mais genérica e, portanto, mais flexível a mudanças.

Nos testes feitos, buscou-se avaliar a questão da performance com relação ao tempo de resposta. Porém, existem outros parâmetros que podem ser observados como, por exemplo: consumo de memória e consumo de CPU. Se o objetivo fosse economizar memória, provavelmente a ativação sob-demanda NO_RETAIN que obteria melhores resultados. Quanto ao consumo de CPU, seria necessário criar novos POAs, variando a política de gerenciamento de threads, para descobrir quais combinações geram os melhores aproveitamentos de CPU.

Como trabalho futuro pretendemos fazer novos testes com o nosso modelo para avaliar o consumo de memória e CPU. Outra melhoria interessante é após o deployment ter sido completado, adicionar a possibilidade de modificar, em tempo de execução, o modelo de ativação de um dado componente. Além disso, pretendemos comparar o desempenho do nosso modelo com o modelo de deployment simplificado apresentado na especificação do CCM.

# Referências


[1] E. Gamma, R. Helm, R.Jonhson, J. Vlissides "Design Patters: Elements of Reusable Object-Oriented Software". Reading, MA: Addison-Wesley, 1995

[2] Siegel, J. "Corba 3 – Fundamentals and Programming" 2nd Edition – OMG Press – 2000

[3] Cornell G., Horstmann C. "Core Java" SunSoft Press, 1997

[4] Stroustrup, Bjarne. "The C++ Programming Language" - Addison-Wesley, 1997

[5] Object Management Group. CORBA Components Volume I. Documento orbos/99-07-01, 1999.

[6] The JavaBeans project web site: http://java.sun.com/beans

[7] Object Management Group. CORBA 2.6 – Capítulo 11 – Portable Object Adapter. Document formal/01-12-49. 2001

[8] D. C. Schmidt, I. Pyarali, C. O'Ryan, N. Wang, V. Kachroo, A. Gokhale, "Applying Optimization Principle Patters in Real-time ORBs, 5th USENIX Conference on OO Technologies and Systems (COOTS'99), San Diego, CA, Maio 1999.

[9] D.C. Schmidt, S. Vinoski, "Object Adapters: Concepts and Terminology", C++ Report Magazine, vol. 11, Novembro/Dezembro 1997.

[10] Ron Zahavi, "Enterprise Application Integration with CORBA Component and Web-Based Solutions", John Wiley & Sons Inc., 1999.

[11] Orfali R., Harkey D., "Java and Corba" 2nd Edition – John Wiley and Sons Inc. 1998.

[12] Kebbal, D., Bernard, G., "Component Search Service and Deployment of Distributed Applications" –Institut National des Télécommunications – Département Informatique – Évry – France – Relátorio Técnico 0-7695-1300, pp 125-134.

[13] TAO:

http://www.cs.wustl.edu/~schmidt/ACE_wrappers/TAO.tar.gz acessado em Outubro de 2002.

[14] C. Szyperski. "Component Software: Beyond Object-Oriented Programming". Addison-Wesley, 1998.

[15]. Hissam, S. A., Moreno, G. A., Stafford, J. A., and Wallnau, K. C. "Packaging Predictable Assembly". First International IFIP/ACM Working Conference on Component Deployment, Junho de 2002, Berlin, Alemanha, publicação LNCS.

[16]. Kebbal, D., Bernard, G. "Component Search Service and Deployment of Distributed Applications". Proc. 3rd International Symposium on Distributed Objects and Applications (DOA'01), Roma, Itália, Setembro de 2001.

[17] Mikic–Rakic, M., Medvidovic, N. "Architecture–Level Support for Software Component Deployment in Resource Constrained Environments". First International IFIP/ACM Working Conference on Component Deployment, Junho de 2002, Berlin, Alemanha, publicação LNCS.

[18] Rudkin, S., Smith, A. "A Scheme for Component Based Service Deployment". Proceedings of Trends in Distributed Systems: Towards a Universal Service Market, Setembro de 2000, Springer-Verlag, pp 68-80

[19] Wang, N., Parameswaran, K., Kircher, M., Schmidt, D. "Applying Reflective Middleware Techniques to Optimize a QoS-enabled CORBA Component Model Implementation". Proceedings of the 24th Annual International Computer Software and Applications Conference (COMPSAC 2000), Taipai, Taiwan, Outubro de 2000.

[20] Wang, Y.M., Chung, P.E., Huang, Y., Yajnik, S., Shin, J., Liang. D. "DCOM and CORBA Side by Side , Step by Step, and Layer by Layer", Setembro de 1997. http://www.research.microsoft.com/~ymwang/papers/HTML/DCOMnCORBA/S.html acessado em Outubro de 2002.

[21] Henning, Michi "Binding, Migration, and Scalability in CORBA" CRC for Distributed Systems Technology (DSTC), University of Queensland, Austrália.

[22] Barros, M. C. B., Madeira E.R.M., Sotoma I. – "A Deployment Model for CORBA Components" – International Symposium on Distributed Object & Applications – DOA '02, Irvine, EUA, Outubro de 2002, Relatório Técnico – Universidade da Califórnia, Irvine, pp 9-12.

[23] Barros, M.C.B., Madeira E.R.M., Sotoma I. – "An Experience on CORBA Component Deployment" – IEEE Sixth International Symposium on Autonomous Decentralized Systems – ISADS 2003, Pisa, Itália, Abril de 2003, pp 319-326.

[24] CORBA Component Model, versão 3.0, formal/2002-06-65 http://www.omg.org/docs/formal/02-06-65.pdf acessado em Julho de 2003.

[25] Crnkovic, I., Hnich, B., Jonsson, T., Kiziltan, Z. – "Specification, Implementation and Deployment of Components" – Communications of the ACM, vol. 45, Outubro de 2003, pp 35-40.

[26] Mico CCM http://www.fpx.de/MicoCCM/ acessado em Julho de 2003.

[27] OpenCCM http://openccm.objectweb.org/index.html acessado em Julho de 2003.

# Apêndice A – Definições utilizadas no Repositório de Componentes

Nesse apêndice estão as definições do *ComponentRepository*, *ComponentDef* e *HomeDef*.

```
interface  ComponentRepository : Repository {

      ComponentDef  create_component (in RepositoryIdid,
                                      in Identifier        name,
                                      in VersionSpec  version,
                                      in Componentdef base_component,
                                      in InterfaceDefSeq
supports_interfaces);

      HomeDef  create_home (in RepositoryId    id,
                            in Identifier      name,
                            in VersionSpec     version,
                            in HomeDef         base_home,
                            in ComponentDef    managed_component,
                            in ValueDef        primary_key);

}; // interface ComponentRepository

interface ComponentDef : InterfaceDef {

      // read/write interfaces
      attribute interfaceDefSeq          supported_interfaces;
      // read interfaces
      readonly attribute     ComponentDef          base_component;
      readonly attribute     ProvidesdefSeq        provides_interfaces;
      readonly attribute     UsesDefSeq            uses_interfaces;
      readonly attribute     EmitsDefSeq           emits_events;
      readonly attribute     PublishesDefSeq       publishes_events;
      readonly attribute     ConsumesDefSeq        consumes_events;
      readonly attribute     boolean               is_basic;
      // write interfaces
```

```
      ProvidesDef create_provides (in RepositoryId   id,
                                   in Identifier     name,
                                   in VersionSpec    version,
                                   in InterfaceDef   interface_type);
      UsesDef  create_uses (in RepositoryId    id,
                            in Identifier      name,
                            in VersionSpec     version,
                            in InterfaceDef    interface_type,
                            in boolean         is_multiple);
      EmitsDef  create_emits (in RepositoryId  id,
                              in Identifier    name,
                              in VersionSpec   version,
                              in ValueDef      value);
      PublishesDef  create_publishes (in RepositoryIdid,
                                      in Identifier  name,
                                      in VersionSpec version,
                                      in ValueDef    value);
      ConsumesDef  create_consumes (in RepositoryId  id,
                                    in Identifier    name,
                                    in VersionSpec   version,
                                    in ValueDef      value);
}; // interface ComponentDef


interface HomeDef : InterfaceDef  {

      // read operations
      readonly attribute HomeDef    base_home;
      readonly attribute ComponentDef     managed_component;
      readonly attribute PrimaryKeyDef    primary_key;
      readonly attribute FactoryDefSeq    factories;
      readonly attribute FinderDefSeq     finders;
      readonly attribute boolean          is_basic;

      // write interfaces
      PrimaryKeyDef  create_primary_key (in RepositoryId   id,
                                         in Identifier     name,
                                         in VersionSpec    version,
```

```
                                        in ValueDef primary_key);
      FactoryDef  create_factory (in RepositoryId          id,
                                  in Identifier      name;
                                  in VersionSpec     version,
                                  in ParDescriptionSeq     params,
                                  in ExceptionDefSeq  exceptions);

      FactoryDef  create_finder  (in RepositoryId          id,
                                  in Identifier      name;
                                  in VersionSpec     version,
                                  in ParDescriptionSeq     params,
                                  in ExceptionDefSeq  exceptions);
}  // interface HomeDef

interface PrimaryKeyDef : Contained {
      // read interface
      boolean  is_a  (in RepositoryId  primary_key_id);
      readonly attribute  ValueDef  primary_key;
}

interface FactoryDef : OperationDef {
}

interface FinderDef : OperationDef {
}
```

# Apêndice B – Formato do Repositório CAI

A Tabela abaixo ilustra a entrada no repositório CAI referente ao caso de uso após a conclusão do deployment.

| CompAssemblyId: 27843568 | |
|---|---|
| CompInstance: | |
| CompInstanceId: | 10001 |
| CompInstanceTag | "LeftSmile" |
| DLLName/CreateFunction | MySmile/create_MySmile |
| Hostname/DirectoryDLLFrom | marumbi:/opt/software/free/MySmile.dll |
| Hostname/DirectoryDLL | gaivota:/install/free/MySmile.dll |
| Hosname/DirectoryPropertyFileFrom | marumbi:/users/init/leftSmileStatus.txt |
| Hostname/DirectoryPropertyFileTo | gaivota:/users/init/leftSmileStatus.txt |
| InterfRepId | IDL:MySmile:1.0 |
| Activation Model / ServantRetentionPolicy | ON_DEMAND / NO_RETAIN |
| CompCategory | Service |
| Hostname | gaivota |
| Reference | referência para a instância de componente LeftSmile (seqüência de números hexadecimais) |
| CompInstance: | |
| CompInstanceId | 10002 |
| CompInstanceTag | "RightSmile" |
| DLLName/CreateFunction | MySmile/create_MySmile |
| Hostname/DirectoryDLLFrom | marumbi:/opt/software/free/MySmile.dll |
| Hostname/DirectoryDLLTo | gaivota:/install/free/MySmile.dll |
| Hostname/DirectoryPropertyFileFrom | marumbi:/users/init/rightSmileStatus.txt |
| Hostname/DirectoryPropertyFileTo | gaivota:/users/init/rightSmileStatus.txt |
| InterfRepId | IDL:MySmile:1.0 |

| | |
|---|---|
| Activation Model / ServantRetentionPolicy | ON_DEMAND / NO_RETAIN |
| CompCategory | Service |
| Hostname | gaivota |
| Reference | referência para a instância de componente RightSmile (seqüência de números hexadecimais) |
| CompInstance: | |
| CompInstanceId | 10003 |
| CompInstanceTag | “HappyButton” |
| DLLName/CreateFunction | MyButton/create_MyButton |
| Hostname/DirectoryDLLFrom | marumbi:/opt/software/free/MyButton.dll |
| Hostname/DirectoryDLLTo | gaivota:/install/free/MyButton.dll |
| Hostname/ DirectoryPropertyFileFrom | marumbi:/users/init/HappyButtonStatus.txt |
| Hostname/DirectoryPropertyFileTo | gaivota:/users/init/HappyButtonStatus.txt |
| InterfRepId | IDL:MyButton:1.0 |
| ActivationModel/ ServantRetentionPolicy | ON_DEMAND / NO_RETAIN |
| CompCategory | Service |
| Hostname | gaivota |
| Reference | referência para a instância de componente HappyButton (seqüência de números hexadecimais) |
| CompInstance: | |
| CompInstanceId | 10004 |
| CompInstanceTag | “SadButton” |
| DLLName/CreateFunction | MyButton/create_MyButton |
| Hostname/DirectoryDLLFrom | marumbi:/opt/software/free/MyButton.dll |
| Hostname/DirectoryDLLTo | gaivota:/install/free/MyButton.dll |
| Hostname/DirectoryPropertyFileFrom | marumbi:/users/init/SadButtonStatus.txt |

| | |
|---|---|
| Hostname/DirectoryPropertyFileTo | gaivota:/users/init/SadButtonStatus.txt |
| InterfRepId | IDL:MySmile:1.0 |
| Activation Model / ServantRetentionPolicy | ON_DEMAND / NO_RETAIN |
| CompCategory | Service |
| Hostname | gaivota |
| Reference | referência para a instância de componente SadButton (seqüência de números hexadecimais) |
| EventConectionInstance: | |
| ConnectionId | 2001 |
| ConnectionTag: | "HappyButtonPressed" |
| ConnectionSideAId | 10003 |
| ConnectionSideARef | referência para Happy Button |
| ConnectionSideZId | 10001 |
| ConnectionSideZRef | referência para Left Smile |
| EventNameA: | ButtonPressedEvent |
| EventNameZ: | ButtonPressedEvent |
| Reference: | referência para a instância do canal de eventos (seqüência de números hexadecimais) |
| EventConectionInstance: | |
| ConnectionId | 2002 |
| ConnectionTag: | "SadButtonPressed" |
| ConnectionSideAId | 10004 |
| ConnectionSideARef | referência para Sad Button |
| ConnectionSideZId | 10001 |
| ConnectionSideZRef | referência para Left Smile |
| EventNameA: | ButtonPressedEvent |

| | |
|---|---|
| EventNameZ: | ButtonPressedEvent |
| Reference: | referência para a instância do canal de eventos (seqüência de números hexadecimais) |
| LinkConectionInstance: | |
| ConnectionId | 3001 |
| ConnectionSideAId | 10001 |
| ConnectionSideARef | referência para Left Smile |
| ConnectionSideZId | 10002 |
| ConnectionSideZRef | referência para Right Smile |
| FacetName | actionToBePerformedAtNeighbour |
| ReceptacleName: | actionToBePerformed |

# Apêndice C – Descrição das classes Gerente de Servant, Localizador de Servant e Ativador de Servant

Esta é a descrição da classe que implementa o Gerente de Servant. Ela serve como classe auxiliar para as classes Localizador de Servant e Ativador de Servant, contendo os métodos em comum utilizados por ambas as classes.

```
class ServantManager_i
{
public:
  // Esse é um typedef usado para fazer o cast do void* obtido
  // ao descobrir a função de ciração dentro da DLL
   typedef PortableServer::Servant
           (*SERVANT_FACTORY) (CORBA::ORB_ptr orb,
                               PortableServer::POA_ptr poa);
  // Construtor
   ServantManager_i (CORBA::ORB_ptr orb);

  // Destrutor
   ~ServantManager_i (void);

  // Obtém um servant para ativação carregando a DLL apropriada
  // e chamando a função para criar o objeto servant
  // O parâmetro <str> é o ObjectId que contém o nome da DLL e da
  // função de criação
  PortableServer::Servant obtain_servant (const char *str,
                                          PortableServer::POA_ptr poa);

  // O servant é destruído e a DLL que estava ligada dinamicamente
  // é fechada
  void destroy_servant (PortableServer::Servant servant,
                        const PortableServer::ObjectId &oid);

  // Função auxiliar para criar o ObjectId contendo o nome
```

```
  // da DLL e da função de criação
  PortableServer::ObjectId_var create_dll_object_id (const char *libname,
                                          const char *factory_function);

 private:

  // referência para o ORB
  CORBA::ORB_var orb_;

  // nome da DLL
  ACE_CString dllname_;

  // nome da função de criação
  ACE_CString create_symbol_;

  // Esse é um objeto HashMap, utilizado para obter um acesso rápido a DLL
  // associada a cada Servant através da chave única ObjectId
  typedef ACE_Hash_Map_Manager_Ex<PortableServer::ObjectId,
                                  ACE_DLL *,
                                  TAO_ObjectId_Hash,
                                  ACE_Equal_To<PortableServer::ObjectId>,
                                  ACE_Null_Mutex>
                                 SERVANT_MAP;

  SERVANT_MAP servant_map_;

};
```

Esta é a descrição da classe que implementa o Localizador de Servant:

```
class ServantLocator_i : public POA_PortableServer::ServantLocator
{
  //  Essa classe define a interface Servant Locator do Gerente de Servant.
  //  É invocada quando o POA tem as políticas USE_SERVANT_MANAGER e
  //  NON_RETAIN.

public:
  // Construtor
```

```
  ServantLocator_i (CORBA::ORB_ptr orb);


  // Esse método é invocado pelo POA sempre que é recebida uma requisição
  // para um objeto não-ativo. Quando o POA é criado com a política
  // NON_RETAIN o Mapa de Objetos Ativos não é mantido, em outras palavras
  // a associação entre o ObjectId e o Servant não é mantida.
  // Portanto, para cada requisição o Servant tem que ser carregado
  // novamente. Note o parâmetro operation, esse parâmetro  indica a
  // operação a ser invocada no Servant. O parâmetro cookie auxilia na
  // marcação/identificação do Servant, e o cookie é util para a destruição
  // do Servant.
  // Esse método é invocado para cada requisição do cliente.
  virtual PortableServer::Servant preinvoke (
          const PortableServer::ObjectId &oid,
          PortableServer::POA_ptr adapter,
          const char *operation,
          PortableServer::ServantLocator::Cookie &the_cookie,
          CORBA::Environment &ACE_TRY_ENV);


  // Esse metodo é invocado sempre que o Servant completa uma requisição
  // Como a interface ServatLocator é usuada quando o POA não mantém
  // o Mapa de Objetos Ativos, é necessario se livrar do Servant depois
  // que a requisição foi processada. O Servant apropriado é destruído
  // através da identificação provida pelo cookie.
  // Também, esse médoto é invocado para cada requisição do cliente.
  virtual void postinvoke (const PortableServer::ObjectId &oid,
                           PortableServer::POA_ptr adapter,
                           const char *operation,
           PortableServer::ServantLocator::Cookie the_cookie,
                           PortableServer::Servant the_servant,
                           CORBA::Environment &ACE_TRY_ENV);

private:
  // O objeto <ServantManager_i> provê métodos úteis tais como:obtenção de
  // um servant utilizando o obejto ACE_DLL, destruição de um servant,
  // extração dos nomes da função de criação e da DLL do ObjtectId
```

```
  ServantManager_i servant_manager_;

};
```

Esta é a descrição da classe que implementa o Ativador de Servant:

```
class ServantActivator_i : public POA_PortableServer::ServantActivator
{
  //   Essa classe associa e desassocia um Servant com um objeto no
  //   Mapa de Objetos Ativos do POA.

public:

  // Inicialização.
  ServantActivator_i (CORBA::ORB_ptr orb);

  // Esse método é invocado por um POA com políticas USE_SERVANT_MANAGER e
  // RETAIN sempre que ele recebe uma requisição para um objeto que não está
  // ativo. Quando um Servant correspondente ao ObjectId não é encontrado
  // no Mapa de Objetos Ativos, o POA obtém o Servant através da classe
  // Gerente de Servant e a seguir o armazena no Mapa.
  virtual PortableServer::Servant incarnate (
         const PortableServer::ObjectId &oid,
                PortableServer::POA_ptr poa,
              CORBA::Environment &ACE_TRY_ENV);

  // Esse método é invocado sempre que o objeto é desativado.
  // Isso ocorre quando o POA é destruído ou quando o objeto é desativado.
  // Quando o POA é destruído, é preciso desativar todos os objetos
  // cujos Servants estão armazenados no Mapa de Objetos Ativos e
  // os Servants.
  virtual void etherealize (const PortableServer::ObjectId &oid,
                            PortableServer::POA_ptr adapter,
                            PortableServer::Servant servant,
                            CORBA::Boolean cleanup_in_progress,
                            CORBA::Boolean remaining_activations,
                            CORBA::Environment &ACE_TRY_ENV);
```

```
  // Função auxiliar para criar o ObjectId contendo o nome
  // da DLL e da função de criação
  PortableServer::ObjectId_var create_dll_object_id (const char *dllname,
                                                     const char *factory_function);

private:
  // O objeto <ServantManager_i> provê métodos úteis tais como:obtenção de
  // um servant utilizando o objeto ACE_DLL, destruição de um servant,
  // extração dos nomes da função de criação e da DLL do ObjtectId
  ServantManager_i servant_manager_;

};
```