Universidade Estadual de Campinas - UNICAMP
Faculdade de Engenharia Elétrica e de Computação

Uma Ferramenta Para Suporte ao Desenvolvimento de Software Orientado a Componente

Autor: Carlos Alexandre Miglinski
Orientador: Prof. Eleri Cardozo, Ph.D.

Dissertação de Mestrado apresentada à Faculdade de Engenharia Elétrica e de Computação como parte dos requisitos para obtenção do título de Mestre em Engenharia Elétrica. Área de concentração: **Engenharia de Computação**.

Banca Examinadora

Ivan Luiz Marques Ricarte, Ph.D. ..... DCA/FEEC/UNICAMP
Maria Beatriz Felgar de Toledo, Ph.D. ..... DSC/IC/UNICAMP
Ricardo Ribeiro Gudwin, Dr. ..... DCA/FEEC/UNICAMP

Outubro de 2003
Campinas, SP - Brasil

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

M588f

Miglinski, Carlos Alexandre
Uma ferramenta para suporte ao desenvolvimento de software orientado a componente / Carlos Alexandre Miglinski. – Campinas, SP: [s.n.], 2003.

Orientador: Eleri Cardozo.
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Java (Linguagem de programação de computador). 2. CORBA (Arquitetura de computador). 3. JavaBeans. 4. XML (Linguagem de marcação de documentos). I. Cardozo, Eleri. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Resumo

Miglinski, Carlos Alexandre, "Uma Ferramenta para Suporte ao Desenvolvimento de *Software* Orientado a Componente". Dissertação de Mestrado - DCA/FEEC/UNICAMP, Campinas, SP. Outubro 2003.

A complexidade dos atuais sistemas de *software* e a necessidade de ferramentas que auxiliem na sua construção motivaram a pesquisa e o desenvolvimento de uma ferramenta de geração automática de código. Isto tornou-se viável graças aos recentes avanços obtidos pela engenharia de *software* principalmente no desenvolvimento de *software* baseado em componentes. O objetivo e meta dessa dissertação é a construção de uma ferramenta para suporte ao desenvolvimento de *software* orientado a componente. Esta ferramenta denomina-se ccmBuilder e vem para complementar o modelo de componentes CM-Tel tornando possível modelar um projeto usando-se o modelo CM-Tel e implementá-lo usando a ferramenta ccmBuilder.

Constatou-se que o uso do conjunto CM-Tel/ccmBuilder facilita o desenvolvimento, pois a modelagem é feita usando ferramentas *CASE* tradicionais no mercado. Já na implementação as partes não funcionais e funcionais são gerados na sua totalidade, ficando somente a parte lógica da aplicação a cargo do desenvolvedor. Obtém-se com isto uma agilidade e maior confiabilidade do *software*.

Palavras-chave: Geração Automática de Código, Componentes, CCM, EJB, XML, Parser XML, Objetos Distribuídos, CORBA, Java.

Abstract

Miglinski, Carlos Alexandre, "A Tool for Component Based Software Development.". Masters Thesis - DCA/FEEC/UNICAMP, Campinas, SP. October 2003.

The complexity of the software systems and the need of tools that help in their construction motivated us the development of an automatic code generation tool. This tool take advantage of the recent advances in software engineering, mainly in the area of component-based software development. The main objective of this master thesis is the construction of a tool that supports the development of component-oriented software. This tool is called ccmBuilder and complements the CM-Tel component model by providing a code generation facility to this model.

It was noticed that the combination CM-Tel/ccmBuilder improves the development process. While the modeling is conducted using tradicional CASE tools. The functional and non-functional parts are generated in their totality, being only the logical part of the aplication left to the developer. As a result speed up in development and more reliable software are achieved.

Keywords: Automated Code Generation, Components, CCM, EJB, Parser XML, Distributed Objects, CORBA,Java.

Para minha esposa Joseane e meus filhos Marcelo e Fernanda

"Um dia me disseram que as nuvens não eram de algodão
Sem querer eles me deram as chaves que abrem esta prisão
Quem ocupa o trono tem culpa, quem oculta o crime, também
Quem duvida da vida tem culpa, quem evita a dúvida, também tem

Somos quem podemos ser
Sonhos que podemos ter"

Engenheiros do Haway

Agradecimentos

Ao Prof. Eleri Cardozo, sou grato pela orientação, dedicação e carinho que me foram ofertados.

Aos meus avós, "Nico" (*in memorian*) e Rosa, por terem me mostrado o valor da perseverança.

Aos meus pais, Alexandre e Marilda, pelo apoio e incentivo.

À minha esposa, Joseane de Carvalho Risardi Miglinski, pela compreenção das minhas ausências.

À minha colega de trabalho, Raquel Santos Schulze, pela inestimável ajuda na leitura da minha dissertação.

Aos companheiros de trabalho, Eliane G. Guimarães, Antônio T. Maffeis.

Aos amigos e colegas do Laboratório de Computação e Automação, pelos valiosos momento de convívio e calorosas discussões, Rossano P. Pinto, Eduardo Nicola F. Zagari, Naur Janzantti Jr , Rodrigo C.M. Prado, Tomas A. C. Badan, Luis F. Faina, Benito T. Giordani e Luiz G. da Silveira Jr.

Ao Departamento de Engenharia de Computação e Automação Industrial da Faculdade de Engenharia Elétrica e de Computação da UNICAMP por ter proporcionado a infra-estrutura necessária à realização deste trabalho.

Ao Centro Regional Universitário de Pinhal (CREUPI), a quem devo minha formação profissional e pelo apôio durante o período em que estive me dedicando ao mestrado.

Lista de Acrônimos

BOA: *Basic Object Adapter*
CCM: *CORBA Components Model*
CM-Tel: *Component Model for Telematic Application*
CIDL: *Component Implementation Definition Language*
CIF: *Component Implementation Framework*
CORBA: *Common Object Request Broker Architecture*
CPU: *Central Processing Unit*
DCOM: *Distributed Component Object Model*
DTD: *Document Type Definition*
EJB: *Enterprise JavaBeans*
HTML: *HyperText Markup Language*
HTTP: *Hypertext Transfer Protocol*
IDL: *Interface Definition Language*
IDL2: *Interface Definition Language* - versão atual
IDL3: *Interface Definition Language* - versão com extensões
J2EE: *Java 2 Platform, Enterprise Edition*
JSP: *JavaServer Pages*
MOF: *MetaObject Facility*
ORB: *Object Request Broker*
POA: *Portable Object Adapter*
QoS: *Quality of Service*
RMI: *Remote Method Invocation*
RM-ODP: *Reference Model for Open Distributed Processing*
UML: *Unified Modeling Language*
URL: *Uniform Resource Locator*
W3C: *World Wide Web Consortium*

XML: *Extensible Markup Language*

XSL: *Extensible Stylesheet Language*

XSLT: *Extensible Stylesheet Language Transformations*

Sumário

Lista de Figuras

Capítulo 1

Introdução

A complexidade dos serviços oferecidos através da Internet tem aumentado rapidamente. Serviços de telecomunicações que no passado demandavam redes especializadas são hoje oferecidos através da Internet. Estes novos serviços utilizam comunicação multimídia, como no caso de videoconferência, vídeo sob demanda e tele-robótica.

Atualmente, a maioria das complexas aplicações distribuídas são construídas sobre uma plataforma de *middleware* como CORBA (*Common Object Request Broker Architecture*) do OMG (*Object Management Group*)[2], DCOM (*Distributed Component Object Model*)[3] da Microsoft ou RMI (*Java Remote Method Invocation*)[4] da Sun Microsystems. A motivação para o emprego de tais plataformas é acelerar o desenvolvimento de aplicações distribuídas.

Modelar, projetar e implementar aplicações distribuídas têm-se revelado atividades bastante complexas. CORBA fornece ao desenvolvedor uma miríade de escolhas e requer um vasto número de detalhes a serem especificados, tornando a tarefa do desenvolvedor extremamente complexa, lenta e pouco eficiente.

Devido a estas dificuldades, tem-se procurado aprimorar a construção de aplicações distribuídas. Com avanços recentes na engenharia de *software*, principalmente no desenvolvimento de *software* baseado em componentes, verificou-se que um número bem pequeno de regras de projeto têm aplicabilidade comum. No escopo de projeto, estas poucas regras poderiam ser realizadas por ferramentas de geração de código. Isto é análogo ao serviço realizado pelos atuais compiladores IDL que geram todos os *stubs* e *skeletons* necessários à distribuição de objetos. Empregando-se componentes, além de gerar *stubs* e *skeletons*, será possível gerar um volume maior do código da aplicação.

Segundo D'Souza e Wills [5], um componente é uma unidade de *software* que pode ser desenvolvido e instalado independentemente, com interfaces explícitas e bem especificadas tanto para os serviços que ele proporciona quanto para os serviços que ele necessita de outros componentes, e pode ser composto com outros componentes para formar uma unidade maior.

O conceito de componentes de *software* não pode ser considerado uma inovação, pois já era conhecida a possibilidade de desenvolver sistemas grandes e complexos, através da interconexão de blocos de construção menores, que tanto poderiam ser desenvolvidos, aproveitados de desenvolvimentos anteriores ou adquiridos de terceiros. Entretanto, graças à evolução das tecnologias envolvidas, somente agora estas facilidades puderam ser efetiva-

mente implementadas.

Os benefícios com a utilização de componentes de *software* incluem criação de *software* em um período menor, com custo de desenvolvimento reduzido, de alta qualidade e de fácil manutenção.

Este novo paradigma, que combina novas tecnologias orientadas a componentes com tecnologias tradicionais de plataforma de *middleware*, provê um melhor suporte à interoperabilidade, portabilidade , extensibilidade e coexistência com sistemas legados.

Os componentes necessitam de padrões para interação e composição, como também de infra-estrutura e serviços padronizados. Portanto, existe a necessidade de definir modelos de componentes com tais padrões e proporcionar implementações deste modelo de componentes de forma a permitir que componentes e infra-estrutura de componentes sejam projetadas, implantadas e instaladas.

A composição de componentes de *software* somente é possível se um modelo de componentes fornecer padrões para a interação entre componentes de diferentes fornecedores e, eventualmente, que foram implementados em diferentes linguagens de programação.

O termo composição, utilizado neste trabalho e na literatura de tecnologias de componentes de *software*, refere-se a descrição de como os sistemas são montados. Através da composição pode-se formar parte, ou uma aplicação completa, combinando as funcionalidades que o componente possui com as funcionalidades fornecidas por outros componentes. Para permitir composição, um componente, além de encapsular a sua implementação, também deve interagir com o seu ambiente de execução através de interfaces bem definidas. Adicionalmente, uma outra característica importante é a possibilidade de "customizar" as propriedades de um determinado componente para uma aplicação, tanto em tempo de instalação quanto dinamicamente em tempo de execução.

1.1 Objetivos e Contribuições do Trabalho

Esta dissertação apresenta uma ferramenta para construção de aplicações aderentes ao modelo de componentes CM-Tel[6]. O modelo de componentes CM-Tel foi inspirado no Modelo de Componentes CORBA (CCM) sendo uma versão simplificada deste com algumas extensões para suporte a aplicações telemáticas, como, por exemplo, suporte a mídias contínuas.

O modelo de componentes CM-Tel permite agilizar o processo de geração de sistemas distribuídos. Tais sistemas requerem:

- desenvolvimento rápido;
- integração de código legado;
- alta confiabilidade do *software*;
- baixo custo no desenvolvimento.

A ferramenta ccmBuilder contribui para atender os requisitos acima, pois permite um aumento na produtividade, graças à geração automática de código. A confiabilidade do

software também aumenta, pois o código é gerado a partir de padrões (*templates*) amplamente testados. Ao se gerar código de forma mais rápida e confiável, diminui-se também seu custo.

Todo o processo de distribuição de componentes também é gerado pelo ccmBuilder. Em suma, o desenvolvedor irá se concentrar nas funções específicas de sua aplicação, deixando a cargo do ccmBuilder toda a geração do código referente à interação e instalação dos componentes da aplicação.

Seria desejável que todo o processo fosse automatizado, no sentido que o desenvolvedor pudesse criar um projeto usando ferramentas CASE aderente ao padrão UML[7] e, a partir deste, a ferramenta ccmBuilder pudesse gerar os componentes, *containers* e arquivos de distribuição. No momento, é necessário que o desenvolvedor transforme manualmente o projeto modelado através de diagramas UML para arquivos XML[8][9] que por sua vez serão submetidos à ferramenta ccmBuilder para a geração de código.

Para que pudéssemos validar a ferramenta ccmBuilder, foi construído um *software* de vídeo-conferência, ccmVF. Através do desenvolvimento deste *software* pode-se comprovar as vantagens que o desenvolvimento orientado a componente oferece.

1.2 Apresentação do Trabalho

O trabalho está dividido da seguinte maneira:

- No Capítulo 2 é apresentada uma discussão sobre os principais modelos de componentes como *Enterprise Java Beans* (EJB) e *CORBA Components Model* (CCM). O modelo CM-Tel é apresentado neste capítulo.

- No Capítulo 3 são caracterizados os principais requisitos para uma ferramenta de geração de código para o modelo de componentes CM-Tel.

- No Capítulo 4 são apresentados os detalhes de implementação da ferramenta de geração automática de código para o modelo CM-Tel.

- No Capítulo 5 são apresentados os aspectos de implementação de um protótipo gerado a partir da ferramenta ccmBuilder, com o propósito de validar alguns conceitos expostos nos capítulos 3 e 4.

- No Capítulo 6 são apresentadas as principais conclusões e contribuições do trabalho, além da indicação de possíveis futuros trabalhos.

Capítulo 2

Modelos de Componentes

Este capítulo apresenta uma visão geral sobre os modelos de componentes Enterprise JavaBeans (J2EE) e CORBA Component Model (CCM). O capítulo apresenta, ainda, o modelo CM-Tel para o qual foi construída a ferramenta de geração de código objeto desta dissertação.

2.1 Introdução ao Enterprise JavaBeans

O modelo de componentes J2EE (Java 2, *Enterprise Edition*) é um dos primeiros modelos de componentes não proprietário e influenciou na especificação do modelo de componentes CORBA (CCM) do qual nosso modelo CM-Tel se originou.

Segundo a referência [1], aplicações J2EE tipicamente consistem de três camadas que são distribuídas em três diferentes localidades: máquina do cliente, máquina servidora J2EE e máquina servidora de base de dados ou máquinas legadas com tarefas secundárias (figura 2.1). Aplicações que utilizam desta forma estendem o modelo cliente servidor de duas camadas colocando um servidor *multithreaded* entre a aplicação cliente e o armazenamento de informações (arquitetura *3-tier*).

2.1.1 Componentes J2EE

Aplicações J2EE[10][11] são construídas a partir de componentes J2EE. Um componente J2EE é uma unidade de *software* funcional autocontido que é construída em uma aplicação J2EE que está relacionada com classes e arquivos e se comunica com outros componentes. A especificação J2EE define os seguintes componentes:

- Aplicações clientes e applets: são componentes que executam no lado do cliente.
- Java Servlet e JavaServer Pages (JSP): são componentes Web e executam no servidor.
- Componentes Enterprise JavaBeans (EJB): são componentes de negócios e executam no servidor.

Componentes J2EE são escritos em linguagem de programação Java[12] e compilados da mesma forma que qualquer programa Java.

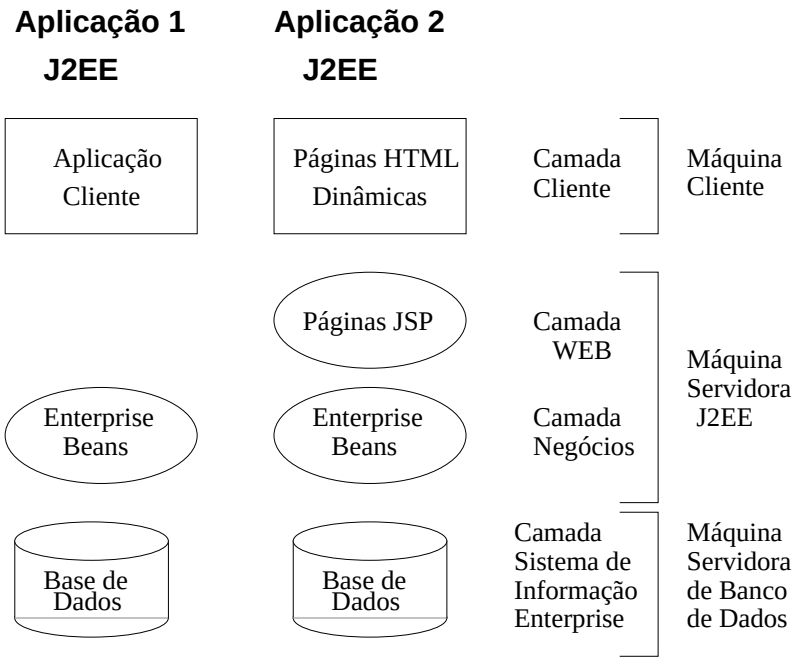

Figura 2.1: Aplicação Multicamada [1].

Principais Vantagens do Modelo EJB

- EJB fornece ao desenvolvedor independência de arquitetura de *hardware* - EJB separa o desenvolvedor da camada de *middleware* dado que o único ambiente que o desenvolvedor EJB vê é o ambiente Java. Também auxilia os fornecedores de servidores/*container* EJB a realizar mudanças na camada de *middleware* sem afetar as aplicações existentes.

- Devido ao EJB ser baseado na tecnologia Java, é garantido tanto para o desenvolvedor quanto ao usuário que seus componentes, uma vez escritos, executam em qualquer lugar (*"Write Once, Run Anywhere"*).

- A especificação EJB associa regras específicas para participantes de projetos cooperarem entre si. Por exemplo, desenvolvedores da lógica da aplicação focam somente na lógica do negócio. Desenvolvedores de aplicações para distribuição se preocupam com as tarefas de distribuição em uma forma simples e portável. O fornecedor de servidores EJB pode cuidar do suporte para serviços complexos de sistemas e tornar disponível um *framework* para um EJB ser executado, sem a assistência do desenvolvedor de aplicações EJB.

- É requerido do fornecedor de *container* EJB fornecer suporte a transações. O desenvolvedor de EJB que escreve a funcionalidade do negócio não precisa se preocupar em iniciar ou terminar uma transação.

- EJB provê transparência para transações distribuídas. Isto significa que uma transação

pode envolver componentes EJB instalados em *containers* executando em máquinas distintas.

- EJB e CORBA são uma combinação natural que se complementam mutuamente. Por exemplo, EJB pode fornecer CORBA/IIOP para um mecanismo de transporte robusto ou clientes CORBA podem acessar componentes EJB como se fossem servidores CORBA.

Componente Enterprise JavaBeans

Um componente EJB é um componente que é executado dentro de um *container* que por sua vez executa dentro de um servidor. Qualquer servidor que possa hospedar um *container* EJB e fornecer os serviços necessários pode ser um servidor EJB.

Um componente EJB é uma classe Java que implementa a lógica do negócio.

Container Enterprise JavaBeans

O *container* EJB é onde o componente EJB "vive". O *container* EJB fornece serviços como transações e gerenciamento de recursos, escalabilidade, mobilidade, persistência e segurança para os componentes EJB. Uma vez que o *container* cuida destas funções, o desenvolvedor pode concentrar-se somente na lógica da aplicação.

Objeto de Acesso e a Interface Remota

O objeto de acesso implementa a interface remota do componente EJB no servidor. A interface remota representa os métodos do negócio do componente EJB. Objetos de acesso e componentes EJB são classes separadas, porém implementando a mesma interface (a interface remota do componente EJB). Entretanto, estas implementações realizam funções distintas. Um componente EJB executa em um *container* EJB dentro de um servidor EJB e implementa a lógica do negócio. O objeto de acesso EJB executa no cliente e, remotamente, executa os métodos do componente EJB. A implementação de um objeto de acesso EJB é criada por uma ferramenta suprida como *container* EJB.

Quando um cliente chama um método em um objeto de acesso EJB, o método do objeto de acesso EJB se comunica com o *container* EJB, solicitando a invocação do mesmo método com os mesmos argumentos supridos pelo cliente. A figura 2.2 demonstra como funciona o sistema cliente/servidor no EJB.

Tipos de Enterprise JavaBeans

Existem dois tipos básicos de componentes EJB, *session beans e entity beans*, que executam diferentes regras em uma aplicação distribuída EJB.

- *Session bean* é uma instância de um EJB associada com um único cliente. *Session bean* tipicamente são transientes e podem ou não participar de transações. Em geral objetos de sessão não sobrevivem à falhas no servidor. Um exemplo de um objeto de sessão pode ser um EJB sendo executado dentro de um servidor Web que serve

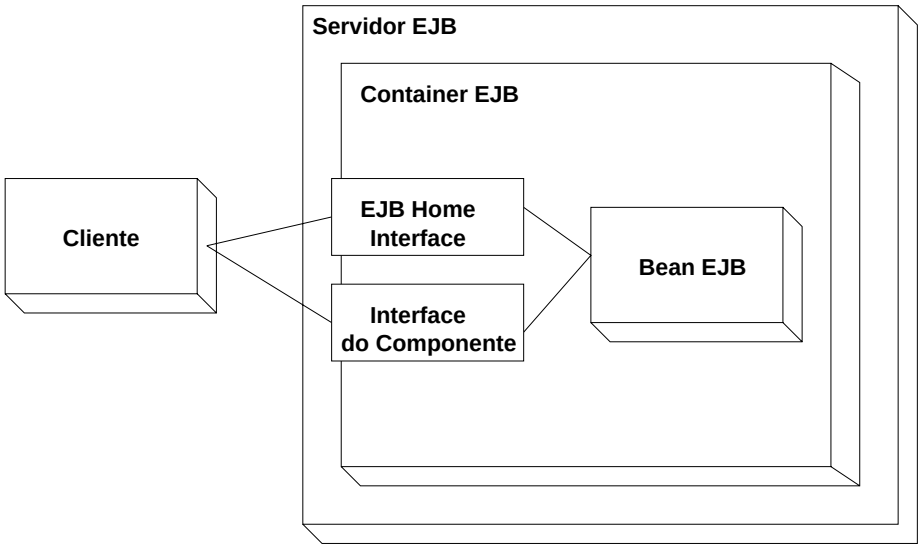

Figura 2.2: Cliente/Servidor no EJB.

páginas HTML em uma sessão HTTP. Quando um usuário abandona a página, o objeto de sessão é destruído.

- *Entity bean* representa informações persistentes armazenadas em uma base de dados. *Entity beans* são associados com transações em banco de dados e podem fornecer acesso a múltiplos usuários. Um *entity bean* é persistente no sentido que sobrevive a falhas no servidor. Um *entity bean* também permite o compartilhamento de informações.

- *Message-driven bean* é um *enterprise bean* que permite que aplicações J2EE processem mensagens assincronamente. Ele atua como um *listener* que recebe mensagens.

  A principal diferença entre *message-driven* e *session/entity* é que o cliente não pode acessar o componente através de uma interface.

Um cliente EJB cria objetos EJB no servidor e os manipula como se fossem objetos locais. Isto torna o desenvolvimento de clientes EJB semelhante ao desenvolvimento de clientes que executam localmente. *Session bean* gerencia informações relacionadas com a interação entre o cliente e o servidor, enquanto *entity bean* representa e manipula dados persistentes no domínio da aplicação.

Interface *home*

Operações envolvendo o ciclo de vida de um componente no lado do servidor são responsabilidade do *container* EJB. O cliente contata o *container* e solicita que um tipo particular de objeto seja criado, e o *container* retorna um objeto EJB que pode ser usado para manipular o novo componente EJB.

Cada classe do componente EJB possui uma interface *home*, que define métodos para criar, inicializar, destruir e localizar instâncias de EJB no servidor. A interface *home* é

um contrato entre o componente EJB e seu *container*, que define construção, destruição e localização de instâncias de EJB.

Uma interface *home* estende a interface *javax.ejb.EJBHome*, que define funcionalidades básicas para uma interface *home*. Todos os métodos nesta interface devem ser compatíveis com RMI. A interface *home* do EJB também define um ou mais método *create()*, cujo nome sempre será *create*, mas com assinaturas diferentes. O valor retornado pelo método *create* é a interface remota para o EJB.

Quando um cliente deseja criar um componente no lado do cliente, ele usa o *Java Naming and Directory Interface (JNDI)* para localizar a interface *home* do componente que deseja.

Uma vez que o cliente possui a interface *home* do EJB que deseja criar, ele chama um dos métodos *create()* da interface *home* para criar o objeto no lado do servidor. A interface *home* no lado do cliente faz uma chamada remota ao *container* EJB no servidor, que, então, cria o componente EJB e retorna um objeto EJB para o cliente. O cliente pode, então, chamar o método do objeto EJB, que irá ser encaminhado ao *container*. O *container* delega a implementação do método para o componente EJB.

Entity beans também tem um método adicional *finder* na interface *home* que localiza o JavaBean persistente, baseado na chave primária do componente.

Descritor de Distribuição

As regras do descritor de distribuição são obtidas de informações declarativas (ex. informações que não são incluídas diretamente no código do *enterprise bean*).

Existem dois tipos básicos no descritor de distribuição:

- Informações estruturais do *enterprise bean*. Informações estruturais descrevem a estrutura de um EJB e declara suas dependências externas. Provêem informações estruturais que não podem ser, em geral, alteradas pois, poderiam prejudicar as funcionalidades do EJB.

- Informações do montador da aplicação descrevem como o EJB no arquivo ejb-jar é composto em uma grande unidade. Prover informações de montagem no descritor de distribuição é opcional para o arquivo ejb-jar produtor. Informações do montador podem ser mudadas sem prejudicar as funcionalidades do EJB.

As informações do descritor de distribuição são escritos em XML e apresentam um DTD para validar o arquivo XML.

O utilitário *deploytool* presente na plataforma J2EE é utilizado para criar pacotes de componentes e para distribuir a aplicação. O utilitário é apresentado em duas versões: uma em linha de comando e a outra utilizando interface gráfica (GUI).

2.2 Introdução ao Modelo de Componentes CORBA

O modelo de componentes CCM (*CORBA Component Model*) do OMG (*Object Management Group*) [13][14] é uma arquitetura para definição, geração, empacotamento e dis-

tribuição de componentes CORBA. CCM adiciona à IDL (*Interface Definition Language*) a capacidade de definir componentes CORBA. CCM também introduz CIDL (*Component Implementation Definition Language*) com a finalidade de descrever detalhes de implementação permitindo um completo *framework* no lado do servidor, que será gerado, montado e distribuído. Apesar do CCM definir um *framework* de componentes focado no lado do servidor, nada impede sua utilização no lado cliente. CCM é dividido em dois níveis distintos: nível básico que componentiza objetos CORBA e um modelo mais rico que possui extensões ao modelo CORBA. A especificação do CCM é estruturada em cinco modelos e um meta modelo.

1. *O modelo abstrato* fornece uma maneira de definir as várias interfaces fornecidas e usadas pelo componente, bem como suas propriedades. Foram adicionadas extensões à IDL para acomodar os novos conceitos introduzidos pelo CCM, especialmente definição de portas (*ports*). O modelo abstrato também permite ao projetista definir gerentes para instâncias de componentes.

2. *O modelo de programação* inclui o framework de implementação de componentes (*Component Implementation Framework* - CIF) que define como as partes funcionais (lógica de aplicação) e não funcional (código gerado automaticamente) devem cooperar. CIF também define uma forma de a instância do componente interagir com o *container* que a hospeda. Este framework conta com o uso de uma Linguagem de Definição e Implementação de Componentes (CIDL).

3. *O modelo de empacotamento* especifica como os componentes e as implementações devem ser agrupadas em pacotes de *software* (empacotadas). Está associado com um conjunto de descritores escrito em XML *(Extended Markup Language)*

4. *O modelo de distribuição* define um processo que permite distribuir uma aplicação de forma simples e automatizada. Este processo usa pacotes de *software* para instalar implementações de componentes, criar instâncias de componentes e interconectá-los.

5. *O modelo de execução* descreve instâncias de componentes em tempo de execução. Este modelo descreve o ambiente de execução para componentes denominado *container*. A característica principal de um *container* é esconder e gerenciar aspectos não funcionais de um componente, como ciclo de vida e persistência.

6. *O meta-modelo* é definido através de esquemas UML (*Unified Modeling Language*) e MOF (*MetaObject Facility*). Este meta-modelo inclui extensões a IDL2 (versão atual da IDL) e o repositório de interfaces bem como o modelo de distribuição de componentes. Este meta-modelo não requer mudanças no UML.

CCM também define um mapeamento para *Enterprise JavaBeans* 1.1, que permite um componente EJB ser visto como um componente CCM e, inversamente, permite um *container* EJB 1.1 acomodado em um ambiente CORBA operar como um *container* CCM.

2.2.1 Definição de Componentes CCM

O modelo abstrato de Componentes CORBA permite ao projetista definir interfaces funcionais do componente usando a linguagem IDL3 do OMG. Estas definições são usadas somente com o propósito de projeto. Para implementar um componente, as definições em IDL3 são mapeadas para IDL2 e, a partir delas, o desenvolvedor irá implementar a parte funcional do componente.

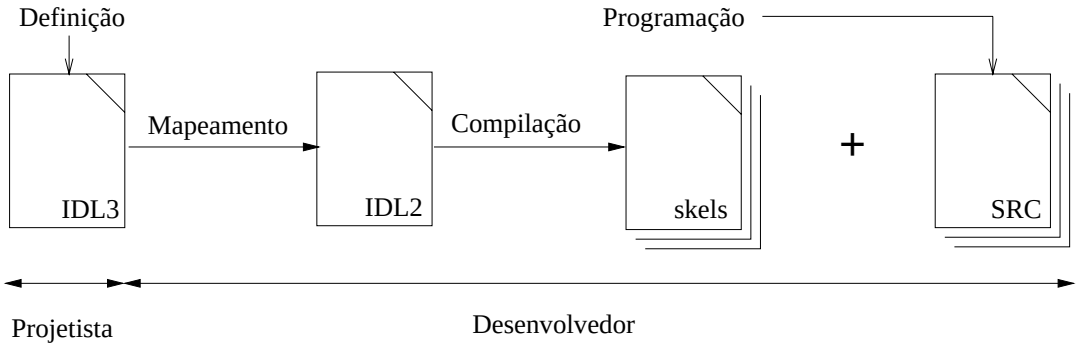

Figura 2.3: Mapeamento de IDL3 para IDL2.

O Tipo Componente

Um tipo componente agrupa definições de atributos e portas (*ports*). Atributos são propriedades configuráveis de um componente e são usados, a princípio, com o propósito de configuração. Portas definem interfaces que são fornecidas ou usadas pelo componente. Portas são novas construções que suportam formas de se "conectar" componentes. Quatro tipos de portas são definidos pelo CCM:

- Facetas: são interfaces fornecidas pelo componente e usadas pelos clientes.
- Receptáculos: estipulam as interfaces de outros componentes que o componente necessita para a realização de suas funções.
- Consumidores de Eventos: são pontos de conexões onde notificações assíncronas (eventos) de um tipo específico são consumidas.
- Emissores de Eventos: são pontos de conexões que emitem eventos de um determinado tipo para um consumidor conectado ou um canal de distribuição de eventos.

Componentes são definidos em IDL3 usando a palavra reservada *component*. Esta palavra reservada representa um novo meta-tipo, que é uma especialização de uma interface. A definição de um tipo componente é muito similar à definição de interface. Como na interface, tipos componentes podem ser definidos usando um relacionamento de herança. Esta interface contém operações relativas a portas e interfaces que derivam de *Components::CCMObject*.

Facetas

Uma faceta (*facet*) representa determinado aspecto funcional de um componente. Uma faceta deve ser especificada por uma interface IDL válida. As facetas não podem ser dissociadas de um componente ao qual pertençam. Muitas cópias da referência de uma faceta podem existir ao mesmo tempo, por exemplo, quando múltiplos clientes usam a mesma faceta de um componente. A palavra reservada *provides* é usada para definir uma faceta. Para permitir a navegação, uma interface base é fornecida: *Components::CCMObject*, que é herdada de *Components::Navigation*. *Componentes::Navigation* provê operações genéricas disponíveis e usadas em qualquer tipo componente. Suas operações principais são: operações de introspecção (*describe_facets()*) e recuperação de faceta (*provide_all_facets(), provide_named_facets() e provide_facet()*). Finalmente, uma operação testa se duas facetas são parte do mesmo componente.

Receptáculos

Um receptáculo permite que um componente aceite uma referência (por exemplo referência de uma faceta, referência de um componente ou referência de um objeto) para que possa usar as operações fornecidas pela referência. O receptáculo é um ponto conceitual de conexão. Esta abstração é materializada em um componente através de um conjunto de operações para agrupar e gerenciar conexões. Receptáculos podem ser de dois tipos: receptáculos simples (somente uma referência pode ser conectada por vez) e receptáculos múltiplos (várias referências podem ser conectadas simultaneamente). Para ambos os casos, receptáculos são definidos usando-se a palavra reservada *uses*. No caso de receptáculos múltiplos, a palavra reservada *multiple* é associada com a palavra reservada *uses*. No contexto de receptáculos múltiplos, uma chave é definida para gerenciar múltiplas conexões. Para cada conexão uma chave é criada assim que a mesma é estabelecida. Esta chave identifica de modo único a conexão e é usada para qualquer operação realizada sobre a mesma. Para ambos os tipos de receptáculos, operações de conexão e desconexão são definidas.

- Conexões - As operações do tipo *connect()* são geradas em parte pela ferramenta de geração de código e em parte implementadas pelo desenvolvedor. Quando se usa receptáculo múltiplo, o desenvolvedor tem que implementar o gerenciamento das referências conectadas. Uma vez que as operações de conexão tenham sido executadas, os receptáculos as mantêm até que uma requisição de desconexão seja recebida.

- Desconexão - As operações do tipo *disconnect()* removem o relacionamento entre a instância do componente e a referência conectada. Em receptáculos simples, a operação retorna a referência que estava conectada ao receptáculo, ou lança a exceção *NoConnection*. Usando-se receptáculos múltiplos, é necessário identificar a chave para finalizar a conexão. Se a chave não for válida a exceção *InvalidConnection* será lançada.

A interface *Components::Receptacles* fornece operações genéricas para conectar/desconectar instâncias de componentes. As operações são aplicadas nas portas cujo nome é passado

como parâmetro. A operação *get_connections()* fornece uma lista de facetas conectadas para um dado receptáculo.

Eventos

O modelo de eventos é um modelo clássico produtor/consumidor. Este modelo é compatível com a especificação de serviços CORBA de eventos e notificação. Para receber eventos, um cliente tem que se subscrever a um originador de eventos. No CCM somente o modelo *push* é utilizado. Operações para processar eventos são definidas na interface do consumidor. Serviços para gerenciar eventos são fornecidos pela implementação do *container*. Existem ainda dois tipos de portas que manipulam eventos: originadores de eventos e consumidores de eventos. Existem dois tipos de originadores de eventos. Um emissor é um originador de eventos que opera na forma de um-para-um, ou seja, somente um consumidor pode ser conectado por vez. Neste modelo não existe canal de eventos, sendo a conexão realizada diretamente entre o produtor e o consumidor. Um publicador é um originador de eventos que opera na forma um-para-muitos, ou seja, muitos consumidores podem ser conectados ao originador de eventos ao mesmo tempo. A subscrição de um consumidor a um tipo de evento é delegada a um canal de eventos fornecido pelo *container*. Um consumidor de eventos permite a uma instância de um componente receber eventos de um determinado tipo. A definição de originadores de eventos é baseada no uso de palavras reservadas *emits* e *publishes*. Neste caso, operações de *connect/disconnect* e *subscribe/unsubscribe* são adicionadas à interface do componente. Quando uma subscrição ocorre, uma chave é gerada para identificar a subscrição do consumidor. Esta chave é necessária para encerrar uma subscrição. Consumidores de eventos são definidos usando-se a palavra reservada *consumes*. Cada porta especifica o tipo de evento (*event type*) produzido ou consumido. Os tipos de eventos são *valuetypes* (objetos por valor) herdados da interface *Components::EventBase*. Os eventos são "empurrados" ao cliente, por isso, os consumidores definem somente uma operação tipo *push()*.

Componente Home

A meta principal de componente tipo *home* é prover o equivalente ao operador *new* em linguagens orientadas a objetos. *Home* é um novo meta-tipo definido em IDL3. Um componente *home* é um gerente para instâncias de um dado componente. Ele gerencia o ciclo de vida do componente usando chaves primárias para indexação. De fato, ele oferece fábricas para instâncias de componentes e operações de busca que usam chave primária. Interfaces *home* são definidas usando-se herança simples do tipo base *Components::CCMHome*. Chaves primárias são identificadores únicos de instância de componentes. As definições são associadas com as palavras reservadas *factory* e *finder*. A interface *home* explícita agrupa operações declaradas pelo projetista. A interface *home* implícita agrupa operações genéricas do componente *home* para este tipo *home*. Quando um *home* não possui chave, a única operação gerada é o método *create*. Nos *homes* com chaves primárias são geradas as operações *create, find e destroy*. Finalmente, operações nas interfaces explícitas têm que ser implementadas pelo desenvolvedor. Operações do *home* são

usadas para encontrar uma instância de um componente. Um *home finder* é obtido fazendo uma chamada a *CORBA::ORB::resolve_initial_references* usando *ComponentHomeFinder* como argumento.

2.2.2 Implementação de Componentes

Modelo de Programação

A meta principal do modelo de programação é descrever aspectos não funcionais para gerar automaticamente esta parte da implementação, deixando a cargo do desenvolvedor somente a codificação da parte funcional do componente. Para que estas duas partes sejam integradas de forma harmônica, o modelo CCM provê um *framework*, o *Component Implementation Framework* (CIF), que descreve como as partes funcionais e não funcionais irão interagir. Para se gerar *skeletons* de um componente, CIF conta com a linguagem *Component Implementation Definition Language* (CIDL), que propõe extensões a IDL2 do CORBA. *Skeletons* de componentes automatizam o gerenciamento de funções básicas como gerenciamento de portas, navegação entre facetas e ciclo de vida do componente.

2.2.3 Empacotamento de Componentes

Para que um componente possa ser distribuído, ele necessita ser empacotado. No modelo CCM um pacote é um arquivo agrupando um componente ou várias implementações de componentes e um conjunto de descritores. Os descritores são especificados através de *Data Type Definition* (DTD). Os descritores são uma boa maneira de se definir a arquitetura de uma aplicação.

Pacotes de *Software*

Um pacote de *software* é constituído de um descritor e um conjunto de arquivos. Estes vários elementos são agrupados em arquivos usando-se o formato "ZIP". No modelo CCM um pacote inclui implementações de componentes CORBA. Um descritor define o conteúdo do pacote. Ele fornece as principais informações sobre o *software* (autor, descrições, interfaces, etc) bem como as descrições de implementação (nome, sistema operacional destino, linguagem de implementação, localização do código, etc). Estas informações são descritas através da linguagem XML.

Descritor de Pacotes de Componentes

Um descritor de pacotes de componentes especifica características dos componentes definidas nas fases de projeto e desenvolvimento. Parte é gerada pelo compilador IDL3 e parte modificada pela ferramenta de empacotamento (alguns elementos como persistência, *threading* e outros). Informações relativas ao tipo de componente, políticas de *threading* e portas fornecidas e requeridas são descritas através da linguagem XML.

Descritor das Funcionalidades

Este descritor é usado através de uma ferramenta de projeto para apresentar informações a respeito do componente. Ele descreve a estrutura do componente: interfaces herdadas, interfaces suportadas, portas, dentre outras informações. As várias interfaces do componente são descritas e referenciadas pelo seu *Repository ID*. Este descritor permite a uma ferramenta conectar componentes em tempo de distribuição para construir uma aplicação em parte ou por completo. Estas informações são descritas através da linguagem XML.

Informações de Distribuição

No momento da distribuição, o descritor de distribuição é usado para escolher o tipo de *container* a ser usado pelo componente e para fornecer para o *container* as informações a respeito das propriedades do componente, como persistência e informações a respeito de políticas de *thread* a serem usadas. Estas informações são descritas através da linguagem XML.

2.2.4 Montagem de Componentes

Um pacote de montagem de componentes fornece uma forma fácil de distribuir componentes. É um modelo/padrão para instanciar um conjunto de componentes e conectá-los. O pacote de montador também é um arquivo "ZIP".

Descritor do Montador

Este descritor especifica a montagem do componente (por exemplo: a descrição do componente, conexões e o particionamento lógico das instâncias). Instâncias dos componentes são conectadas usando portas complementares. O descritor referencia o arquivo de cada componente e especifica suas conexões através de asserções como *connectinterface* e *connectevent* (que estão contidas na declaração *connection*). Isto representa a informação básica do processo de distribuição.

Descritor de Propriedades

Define os parâmetros de configuração do componente e do *home*. É usado para configurar uma instância do componente ou uma instância de um *home*, configurando seus atributos. Fornece propriedades padrões que podem ser substituídas pelo usuário. O descritor de propriedades pode conter quantas entradas forem necessárias. Estas entradas podem ser de três tipos: *simple* para configurar um atributo do tipo simples; *sequence* para configurar um atributo definido como uma seqüência de determinado tipo; e *struct* para configurar atributos compostos de múltiplos tipos.

2.2.5 Distribuição de Componentes

Modelo de Distribuição

O modelo de distribuição é outra contribuição importante do modelo CCM. Ele fornece uma forma de automatizar o espalhamento e instanciação de aplicações distribuídas. Ele contribui para aumentar o reuso de *software* numa forma fácil de uso e integração de componentes existentes.

O processo de distribuição é executado por uma ferramenta fornecida por uma plataforma CCM. Esta ferramenta distribui componentes e monta-os em um computador de destino. Esta aplicação de distribuição é um cliente que usa serviços oferecidos pelos objetos existentes em um sítio de execução para instalar, instanciar e configurar componentes e conexões. Os cinco passos básicos de distribuição são:

1. Definir e escolher um sítio de execução,
2. Instalar implementações onde forem requeridas,
3. Instanciar servidores e *container*,
4. Instanciar *home* e componentes, e
5. Conectar e configurar componentes.

2.2.6 Execução de Componentes

Modelo de Execução

Um *container* é um ambiente de execução para instâncias de componentes CORBA. Ele fornece recursos como memória e CPU, bem como recursos de alto nível como serviços de suporte. Instâncias de componentes são gerenciadas por um *container*. Uma instância de um componente não pode existir sem um *container* para suportá-la. Além disso, um tipo de *container* pode somente hospedar um único tipo de componente para o qual foi designado. Duas categorias de *containers* foram definidas: transientes e persistentes.

2.3 Introdução ao Modelo de Componentes CM-Tel

Do modelo CCM derivou-se um modelo de componentes mais simplificado, chamado CM-Tel. CM-Tel mantém as principais idéias do modelo CCM e adiciona alguns serviços, como suporte a mídias contínuas (*Streams*). O modelo proposto é similar ao CCM nos seguintes aspectos:

- Componentes podem definir propriedades;
- Componentes podem produzir e consumir eventos;
- Componentes executam em *container* e são criados a partir de fábricas (*homes*).

Contudo, algumas diferenças devem ser mencionadas:

- Componentes são especificados em XML;
- A IDL é preservada, com isto a arquitetura de suporte para o modelo de componentes pode ser construída com implementações CORBA atuais;
- Componentes podem manipular fluxos de mídia contínua;
- Uma estrutura simplificada para o *container* foi adotada.

Os elementos principais dos componentes CM-Tel são descritos a seguir:

- Propriedades: definem um conjunto de variáveis de estado usualmente relacionadas com o estado e configuração do componente. Propriedades podem ser inspecionadas e, caso haja privilégio, alteradas por outro componente;
- Produtores e consumidores de eventos: suportam a troca de mensagens de notificação de forma assíncrona com outros componentes;
- Facetas: como no modelo CCM, representam os aspectos funcionais dos componentes;
- Receptáculos: como no modelo CCM, permitem armazenar referências de componentes e objetos que um componente requer;
- Produtores e consumidores de fluxo de mídias: suportam a troca de fluxo de mídia com outros componentes.

Os elementos acima listados podem ser acessados através das interfaces do componente especificadas na linguagem IDL convencional.

Propriedades permitem configurações dinâmicas do componente. Por exemplo, um componente apresentando vídeo pode definir propriedades como formato da mídia, taxa de quadros, tamanho dos quadros e o protocolo de transporte.

Eventos são implementados de acordo com o modelo *push*, onde componentes que produzem eventos tomam a ação de notificar componentes que atuam como consumidores do mesmo evento. CM-Tel utiliza o mesmo modelo proposto pelo CCM: emissor-consumidor (*emitter-consumer*) e publicador-consumidor (*publisher-consumer*). No modelo emissor-consumidor um relacionamento um-a-um conecta um objeto produtor a um único objeto consumidor. No modelo publicador-consumidor um relacionamento um-para-muitos conecta um objeto produtor a zero ou mais objetos consumidores. Neste modelo um canal de eventos faz a intermediação. O publicador envia seu evento ao canal de eventos que distribui este evento para cada consumidor conectado.

Fluxos de mídia contínua foram implementados da mesma forma que eventos. O CM-Tel define dois esquemas para transmissão de mídias contínuas: transmissor-apresentador (*transmitter-player*) e difusor-apresentador (*broadcaster-player*). O transmissor permite a conexão de um único apresentador de mídia. O difusor permite a conexão de zero ou mais apresentadores de mídia. No primeiro caso um fluxo *unicast* será estabelecido, enquanto no segundo caso, um fluxo *multicast* será estabelecido.

Definição do Componente em XML

Um componente no CM-Tel é especificado em XML através de algumas marcações (*tags* XML), que permitem a definição de interfaces do componente bem como suas propriedades, evitando com isso o uso de IDLs estendidas para acomodar as novas construções necessárias para se especificar um componente. As marcações XML são:

- *componentfeatures* - nome do componente e seu identificador no repositório de interfaces (*Repository ID*);
- *ports* - composto das marcações a seguir:
  - *provides* - facetas que representam os aspectos funcionais dos componentes;
  - *uses* - receptáculos que provêem conexões para o uso de referências (tipicamente referências de facetas);
  - *transmits* - produtor de fluxo *unicast* de áudio/vídeo;
  - *broadcasts* - produtor de fluxo *multicast* de áudio/vídeo;
  - *plays* - consumidor de fluxo de áudio/vídeo;
  - *emits* - produtores de eventos na forma um-para-um;
  - *publishes* - produtores de eventos na forma um-para-muitos;
  - *consumes* - consumidores de eventos de um determinado tipo;
- *interface* provêem interfaces CORBA padrão;
- *properties* - definição de propriedades.

A partir das marcações acima relacionadas podemos especificar um componente em XML e submeter esta especificação a uma ferramenta de geração de código que irá produzir classes em Java, arquivos IDLs e um arquivo XML aderente ao padrão *ant*[15][16] para compilação. Com isso, o desenvolvedor não necessita codificar a parte não-funcional da aplicação, concentrando-se somente na lógica realizada pelo componente.

XML

XML[17] é uma linguagem de marcação para documentos contendo informações estruturadas. Informações estruturadas contêm tanto conteúdo (palavras, figuras, etc.) quanto regras que indicam como o conteúdo deve ser interpretado (por exemplo, conteúdo de um cabeçalho possui significado diferente de um conteúdo em uma nota de rodapé).

Uma linguagem de marcação é um mecanismo para identificar estruturas em um documento. A especificação XML define uma forma padronizada de adicionar marcações ao documento.

XML fornece uma forma de definir marcações (*tags*) e o relacionamento estrutural entre elas. Não existe um conjunto de marcações pré-definidas. Toda a semântica de um documento XML deverá ser definida pela aplicação que a processa.

Na linguagem HTML, tanto a semântica das marcações quanto o conjunto das marcações são fixas. O W3C, em conjunto com os vendedores de navegadores e a comunidade WWW, trabalha constantemente para ampliar a definição do HTML e viabilizar novas marcações para acomodar as novas tecnologias emergentes na *Web*.

HTML teve e tem um papel muito importante na disponibilização de informações na *Web*. Contudo, HTML possui algumas limitações. Com a chegada de XML novas características foram introduzidas, garantindo uma maior flexibilidade, pois os documentos em XML são autodescritivos. Um exemplo de aplicabilidade do XML é o seu uso em ferramentas de busca, que permite critérios de seleção mais refinados. XML pode ser utilizado em navegadores que tenham a habilidade de entender as marcações XML, utilizando *eXtensible Stylesheet Language* (XSL) para formatar os documentos. Também é possível o uso do XML em transações *business-to-business*, porém isso requer um mecanismo que leia e interprete o documento XML extraindo seus dados.

Cada documento XML consiste de elementos específicos para aquele tipo de documento. A figura 2.4 mostra a estrutura de um elemento XML. Um elemento contém uma marca (*tag*) de início e uma marca de finalização e o conteúdo entre as marcas.

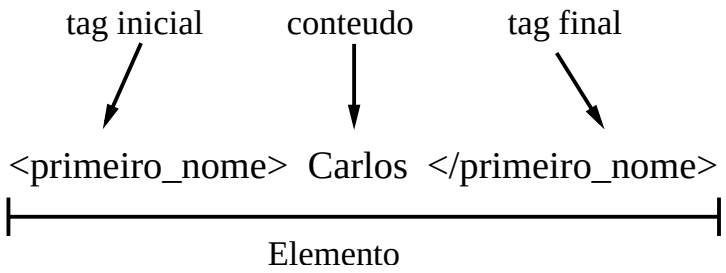

Figura 2.4: Estrutura de um elemento XML.

Qualquer elemento pode conter atributos. Atributos são definidos dentro da marca de início do elemento como se pode ver no exemplo a seguir.

```
<primeiro_nome sexo = "masculino"> Carlos </primeiro_nome>
               |-----------------|
                    Atributo
```

Cada atributo é um par nome-valor. O valor do atributo deve ser usado entre aspas duplas (").

DTD

Um Documento de Definição de Tipo - DTD (*Document Type Definition*) deve acompanhar um documento XML para validá-lo. O DTD contém a estrutura de um documento XML e todas as regras a respeito dos relacionamentos entre elementos e regras particulares para os elementos. O DTD expressa a hierarquia e o aninhamento dos elementos dentro da estrutura.

Alguns princípios básicos serão ilustrados a seguir:

- Um elemento é definido no seguinte formato <!ELEMENT nome do elemento >.
- Um elemento pode conter subelementos que serão colocados entre parênteses e listados após o nome do elemento.
- Cada subelemento deve obedecer as seguintes regras:
  - sem caractere especial - deve ocorrer somente uma vez;
  - asterisco (*) - ocorre zero ou mais vezes;
  - interrogação (?) - pode ocorrer zero ou uma vez;
  - mais (+) - deve ocorrer pelo menos uma vez ou mais vezes.

Os atributos são definidos em uma lista usando o seguinte formato <!ATTLIST>. Um atributo é usado para adicionar significado a uma marcação em particular e deve ser definido no DTD.

```
<!ELEMENT catalogo (nome,item*)>
  <!ATTLIST catalogo estacoes
      (inverno|primavera|verao|outono) #REQUIRED>
 ...
```

No exemplo acima, o catálogo possui um atributo estações que pode ter os valores inverno, primavera, verão ou outono. Neste caso em particular, um valor é obrigatório devido à palavra reservada *#REQUIRED*. *#PCDATA* (*Parsed Character Data*) indica que os dados serão do tipo caractere (texto).

2.4 Resumo do Capítulo

Neste capítulo foram apresentados de forma resumida os principais modelos de componentes não proprietários que nortearam nossa pesquisa, como, *Enterprise JavaBeans*, *CORBA Component Model* e o modelo CM-Tel que originou a ferramenta ccmBuilder objeto desta dissertação. No capítulo seguinte iremos abordar a arquitetura de suporte do modelo CM-Tel, a implementação das interfaces dos componentes, o *container* e o modelo de distribuição.

Capítulo 3

Projeto da Ferramenta ccmBuilder

Este capítulo descreve a arquitetura de suporte do modelo de componentes CM-Tel, seus elementos funcionais e a estratégia de geração de código para este modelo através da ferramenta ccmBuilder.

3.1 Arquitetura de Suporte do Modelo CM-Tel

Arquitetura de suporte do modelo de componentes CM-Tel segue um padrão de arquitetura de n-camadas sugerido por Szyperski[18][19] e representado na figura 3.1. A idéia é posicionar os componentes na primeira camada e adicionar quantas camadas forem necessárias para suportar a integração entre os componentes. Um componente expõe um conjunto de interfaces com operações para manipular seus métodos, suas propriedades, eventos e fluxos de mídia. Operações nas interfaces são fornecidas através de objetos CORBA. Com exceção dos métodos contendo a lógica da aplicação, todos os objetos restantes que suportam as interfaces dos componentes são gerados pela ferramenta ccmBuilder a partir da especificação do componente.

A segunda camada é o *framework* de componentes que irá fornecer toda a infra-estrutura necessária para o componente posicionado na primeira camada. Por exemplo, um componente que possui uma interface que difunde eventos (*Publisher*) irá necessitar de um serviço que realize a difusão (canal de eventos) sendo que este serviço é disponibilizado no *framework* de componentes. Os serviços suportados pelo *framework* de componentes são:

- Fábricas para instanciar os componentes;
- Localizadores (*finders*) que irão retornar a referência das interfaces dos componentes;
- Facilidades para manipular as propriedades de componentes;
- Infra-estrutura para a interconexão de componentes.

As facilidades para a interconexão de componentes são fornecidas pelos serviços CORBA correspondentes. Os seguintes serviços CORBA são fornecidos na segunda camada: Serviço de Propriedades, Serviço de Eventos e Serviço de *Stream* de Áudio/Vídeo (*A/VStreams*).

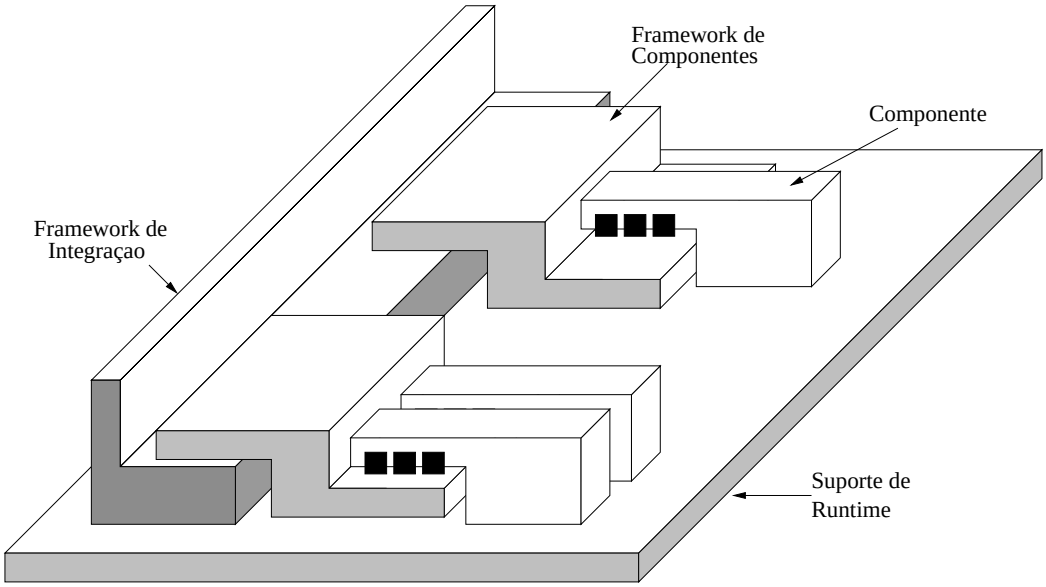

Figura 3.1: Arquitetura de 3 camadas para suportar o modelo de componentes.

A segunda camada consiste de várias bibliotecas (arquivos jar), onde cada serviço CORBA está disponível, ou seja, todas as classes necessárias para a implementação destes serviços. Adicionalmente, uma biblioteca gerada para cada componente armazena a implementação das fábricas dos componentes e seus localizadores.

Finalmente, a terceira camada consiste de um *framework* de integração fornecendo funcionalidade de interoperação entre as outras camadas. Um ORB CORBA irá realizar estas funções de interoperabilidade.

Diferentemente do modelo CCM, CM-Tel define um DTD (*Document Type Definition*) para especificar componentes CORBA em XML. A especificação de um componente começa com o elemento raiz *corbacomponent*. Em seguida o nome do componente é definido no elemento *componentfeatures*. O elemento *componentfeatures* tem um elemento filho importante, chamado de porta (*port*). O elemento porta irá especificar as interfaces do componente. A seguir, é mostrado um fragmento de um arquivo XML utilizado para especificar um componente segundo o modelo CM-Tel.

```
<?xml version="1.0"?>
<!DOCTYPE corbacomponent SYSTEM "ccm.dtd">
<corbacomponent>
  <componentfeatures name = "ccmVideoFone"/>
```

Em seguida iremos detalhar a implementação das interfaces de componentes utilizados neste trabalho.

3.2 Implementação das Interfaces de Componentes

As interfaces de componentes são referenciadas como portas, e são dividas em três grandes classes: interfaces operacionais, interfaces de sinalização e interfaces de fluxos de mídias, tal como definidas no *Reference Model for Open Distributed Processing* (RM-ODP). O nosso modelo apresenta oito tipos de portas: facetas, receptáculos, originadores de eventos, consumidores de eventos, produtores de fluxo de mídia, consumidores de fluxo de mídia, interface equivalente e elementos de configuração (*propertyset*).

Dentro deste modelo, o suporte às interfaces possui um padrão onde uma fábrica instancia o *servant* CORBA que implementa a interface. O *servant* poderá usar serviços CORBA e poderá interagir com objetos da aplicação através de métodos locais para a interação com a porção da aplicação executando no mesmo espaço de endereçamento do *servant* (*container*) . A figura 3.2 exemplifica o modelo de uma interface.

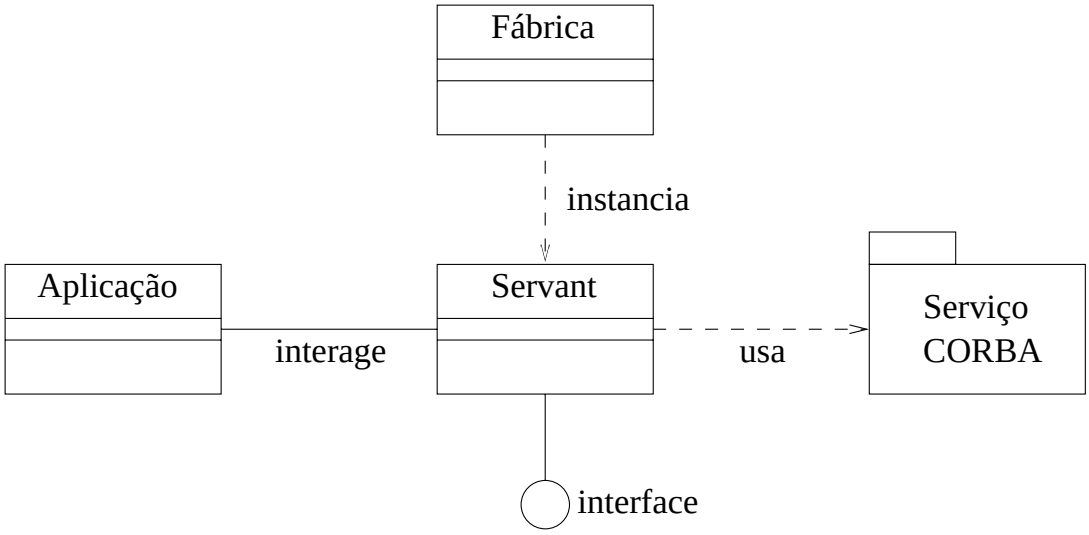

Figura 3.2: Arquitetura de uma interface.

A ferramenta ccmBuilder gera várias classes Java, interfaces IDL e um arquivo XML que será utilizado pela ferramenta Ant[15][16]. Na figura 3.3 podemos notar três tipos de arquivos: IDL, Java e XML.

Os arquivos gerados em IDL são:

- interface equivalente - define operações comuns a todos os componentes, como por exemplo, obter a referência das demais portas. Esta interface é a própria referência do componente;
- interface das portas - facetas, receptáculos, originadores de eventos, consumidores de eventos, produtores de fluxo de mídia e consumidores de fluxo de mídia. Estas interfaces viabilizam a composição entre componentes;
- interface dos elementos de configuração - somente serão gerados caso o componente defina propriedades. Através desta interface torna-se possível acessar e configurar as

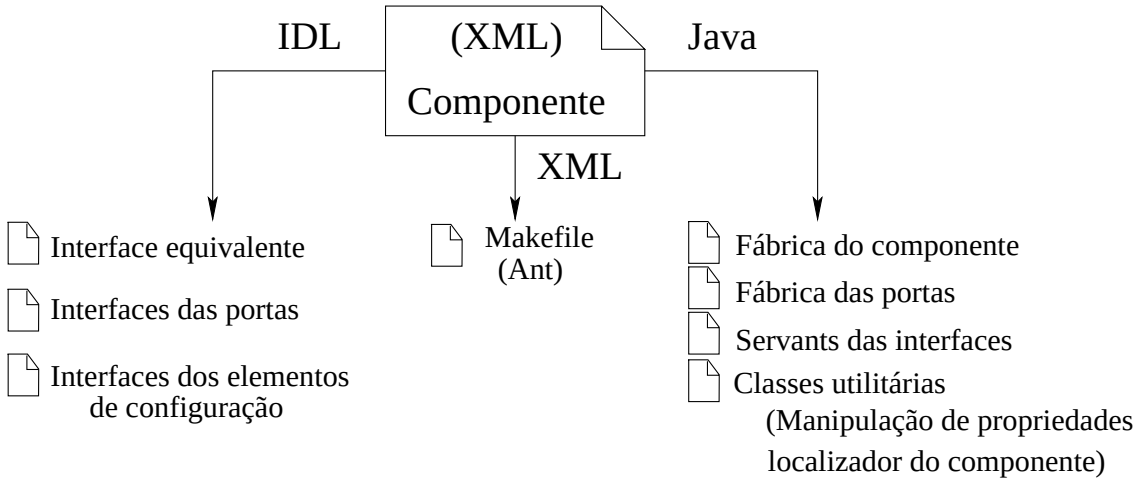

Figura 3.3: Arquivos gerados pelo ccmBuilder para um componente.

propriedades declaradas para o componente.

Os arquivos gerados em Java consistem de:

- fábrica do componente - a fábrica do componente é responsável pelo ciclo de vida do componente através de operações de criação, destruição e o registro do componente na interface equivalente;
- *servants* das interfaces IDL - implementam os métodos das portas;
- fábricas das portas - instanciam os *servants* das portas de um componente;
- classes utilitárias - são utilizadas pelo cliente do componente. O localizador do componente descobre a referência do componente através de uma chave. Os manipuladores de propriedades são classes que permitem a manipulação de propriedades (leitura e alteração dos valores das propriedades).

Os códigos gerados pela ferramenta ccmBuilder obedecem a uma regra de nomenclatura que consiste na obtenção do nome do componente e nome da porta a partir de um arquivo XML. Os códigos são compostos de nome do componente mais tipo de operação a ser realizada, ou nome da porta mais o tipo de ação que a porta irá realizar. Observemos o exemplo a seguir para uma melhor compreensão:

```
<componentfeatures
   name  = "VideoTransmitter"
   repid = "IDL:VideoTransmitter:1:0">

  <ports>
    <!-- video transmitter port -->
    <transmits
```

```
        transmitsname = "videoTr"
        mediatype = "video">
    </transmits>
  </ports>
</componentfeatures>
```

O arquivo XML anteriormente ilustrado especifica um componente que irá implementar uma porta originadora de vídeo. O nome do componente ilustrado é *VideoTransmitter* que será composto com *CCM.idl*, *Factory.java*, *Finder.java*, *Home.idl*, *HomeServant.idl*, *Servant.java*. Da mesma forma ocorre com o nome da porta que neste exemplo é videoTr e será composto com *Factory.java* e *<tipo da porta>Servant.java*. Por exemplo, a fábrica do componente para este exemplo seria *VideoTransmitterFactory.java* e a implementação da porta teria o nome de *videoTrTransmitterServant.java*.

A seguir serão detalhadas as interfaces (portas) propostas neste modelo.

Interface Equivalente

A interface equivalente é uma interface especial que define operações comuns a todos os componentes, como por exemplo obter a referência de uma porta (*provide<nome da porta>*), obter uma chave primária (*get_primary_key*), obter o *home* de um componente (*get_ccm_home*) e obter um indicativo de que a configuração esta completa (*configuration_complete*). A fábrica da interface equivalente é a própria fábrica do componente. A figura 3.4 ilustra a arquitetura de uma interface equivalente.

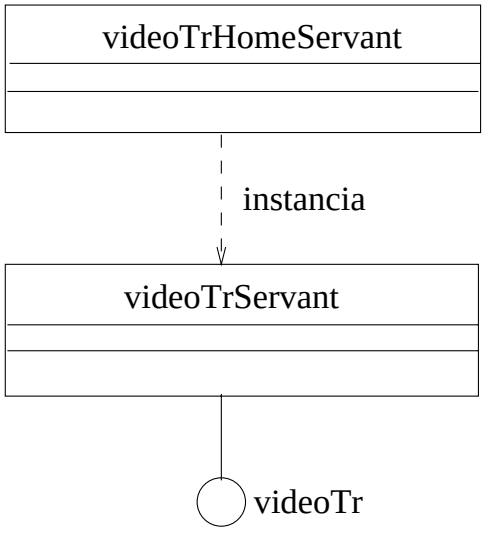

Figura 3.4: Arquitetura da interface equivalente.

A seguir, pode-se observar um arquivo IDL gerado pelo ccmBuilder onde a interface equivalente disponibiliza uma forma de acesso à porta transmissora de vídeo.

```
// Code generated by ccmBuilder
```

```
// Sun Aug 04 17:42:48 PDT 2002

// the component's equivalent facet
interface VideoTransmitter: Components::CCMObject{

    // ports
    videoTrTransmitter provide_videoTr();
};
```

Porta *PropertySet*

A porta *PropertySet* é um elemento de configuração, que permite que determinadas propriedades possam ser configuradas em tempo de execução. A porta de propriedades é especificada em XML e pode ser composta de três tipos básicos: *simple, sequence* e *struct*. O tipo *simple* irá especificar uma propriedade de um determinado tipo, como por exemplo:

```
<simple name = "Enabled" type = "boolean" rigths = "normal">
```

Neste exemplo estamos criando uma propriedade chamada *Enabled* do tipo boleana e com direito de escrita e leitura (a opção *rights* pode assumir os valores *normal* para leitura e escrita ou *readonly* somente para leitura).

O tipo *sequence* especifica uma propriedade que consiste de uma seqüência de um determinado tipo, como por exemplo:

```
<sequence name = "RGB_Cor" type = "sequencia de cores"
rights = "normal">
    <simple name = "red"   type = "float"> <value> 0.0</value> </simple>
    <simple name = "green" type = "float"> <value>10.0</value> </simple>
    <simple name = "blue"  type = "float"> <value>20.0</value> </simple>
</sequence>
```

Como podemos ver no exemplo anterior, o arquivo XML especifica uma propriedade chamada RGB_Cor, que consiste em uma seqüência de números em ponto flutuante, com direito de leitura e escrita.

O tipo *struct* irá especificar uma estrutura que poderá ser composta de vários tipos *simple*. A seguir podemos observar um fragmento de um arquivo XML exemplificando o tipo *struct*.

```
<struct name="Estat" type="EstatStruct" rights="normal">
  <description>Estatisticas</description>
  <simple name="uso" type="float"><value>0.0</value></simple>
  <simple name="erros" type="short"><value>0</value></simple>
  <simple name="arquivos" type="string"><value>log.dat</value></simple>
</struct>
```

Neste exemplo, foi especificada uma estrutura chamada *Estat* composta de um tipo *EstatStruct* agregando um tipo *float* (*uso*), um tipo *short* (*erros*) e um tipo *string* (*arquivos*). A marcação <*description*> permite uma breve descrição da estrutura que esta sendo gerada.

Um exemplo completo de uma especificação em XML para uma propriedade do tipo *simple* é apresentado a seguir.

```
<componentfeatures
   name = "ChatInterface"
   repid = "IDL:ChatInterface:1.0">

  <properties propertiesname = "ChatIntProps">
   <simple name = "Enabled" type = "boolean"  rights = "normal">
     <description>Enables the chat interface when set to true</description>
     <value>false</value>
     </simple>
  </properties>
</componentfeatures>
```

A figura 3.5 exemplifica a arquitetura de uma porta de propriedades (*PropertySet*) mostrando seu processo de criação. A fábrica da porta (*ChatIntPropsFactory*) utiliza a interface *PropertySetFactory* para instanciar a interface *PropertySetFactory*, ambas do serviço de propriedades CORBA. Na instanciação são passadas as propriedades iniciais definidas para o componente obtidas na classe *InitialPropertySet*. A interface *PropertySet* é a porta de propriedades oferecida pelo componente. Para manipular as propriedades a aplicação necessita de métodos *get/set* fornecidos pela classe *ChatIntProperties*.

A seguir apresentaremos um fragmento do código Java gerado para manipular as propriedades. São criados dois métodos *getEnabled* e *setEnabled*; o primeiro permite obter o valor da propriedade, enquanto o segundo possibilita alterar o valor da propriedade.

```
public class ChatInterfaceProperties {
...
  boolean  getEnabled ()
    throws PropertyNotFound,
    InvalidPropertyName {
    try {
      org.omg.CORBA.Any value;
      value = ps.get_property_value("Enabled");
      return (value.extract_boolean());
    }
   ...
  }
```

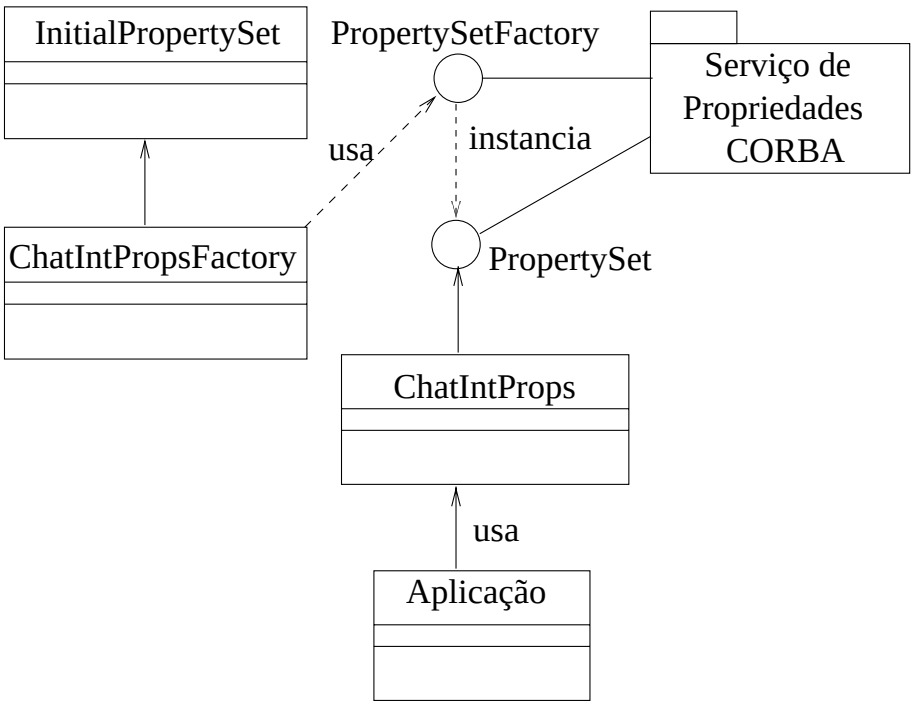

Figura 3.5: Arquitetura da interface PropertyService.

```
  void setEnabled (boolean _value)
    {
    org.omg.CORBA.Any value =
       ORB.init().create_any();
    value.insert_boolean((boolean) _value);
    ps.define_property("Enabled", value);
  }
}
```

Porta *Transmitter*

A porta *Transmitter* é uma porta produtora de fluxo de mídia, que permite a transmissão de áudio/vídeo para uma porta receptora definida em outro componente.

A seguir é apresentado um fragmento de código XML que ilustra as marcações (*tags*) necessárias para representar uma porta *transmitter*. O atributo *transmitsname* irá fornecer o nome da porta, enquanto o atributo *mediatype* indica o tipo de mídia que será transmitida.

```
<ports>
  <!-- video transmitter port -->
  <transmits
     transmitsname = "videoTr"
```

```
     mediatype = "video">
  </transmits>
</ports>
```

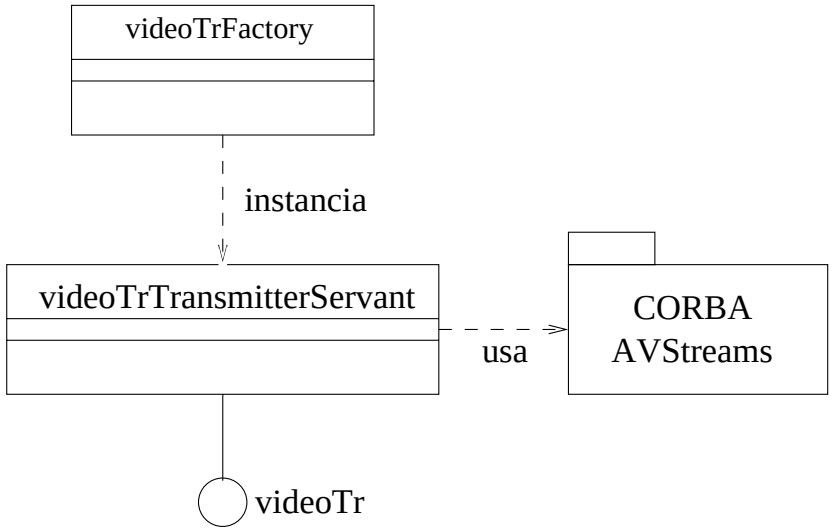

Figura 3.6: Arquitetura da interface transmitter.

O processo de criação da porta *Transmitter* pode ser observado na figura 3.6. A fábrica da porta (*videoTrFactory*) instancia o *servant* (*videoTrTransmitterServant*) que implementa a porta e desta forma disponibiliza a porta de transmissão de vídeo (*videoTr*). A figura ilustra ainda o uso do serviço *AVStreams* que viabiliza a captura e transmissão da mídia selecionada.

A seguir, pode-se observar um arquivo IDL gerado pela ferramenta ccmBuilder para uma porta *transmitter*. O arquivo gerado tem a finalidade de realizar uma conexão (*connect_videoTr*) de uma porta consumidora ou encerrar esta conexão (*disconnect_videoTr*).

```
interface videoTrTransmitter : Video::F_video {
  void connect_videoTr ( in Video::video_Consumer peer)
       raises (AVStreams_Full::alreadyConnected);
  void disconnect_videoTr()
       raises (AVStreams_Full::notConnected);
};
```

Porta *Broadcaster*

A porta *Broadcaster* é uma interface produtora de fluxo de mídia, que permite transmitir áudio/vídeo por difusão. A principal diferença entre a porta *transmitter* e a porta *broadcaster* é que a porta *broadcaster* pode originar mídia para mais de um receptor. A porta *broadcaster* deve ser especificada em XML conforme ilustrado a seguir.

```
<componentfeatures
   name  = "VideoBroadcaster"
   repid = "IDL:VideoBroadcaster:1:0">

  <ports>
    <!-- video broadcaster port -->
    <broadcasts
       broadcastsname = "videoBr"
       mediatype = "video">
    </broadcasts>
  </ports>
</componentfeatures>
```

A figura 3.7 exemplifica a arquitetura de uma porta *broadcaster* mostrando seu processo de criação. A fábrica da porta (*videoBrFactory*) instancia o *servant* (*videoBrBroadcaster-Servant*) que implementa e disponibiliza a porta (*videoBr*). Pode se observar também, nesta figura, que o *servant* utiliza o serviço CORBA *AVStreams*.

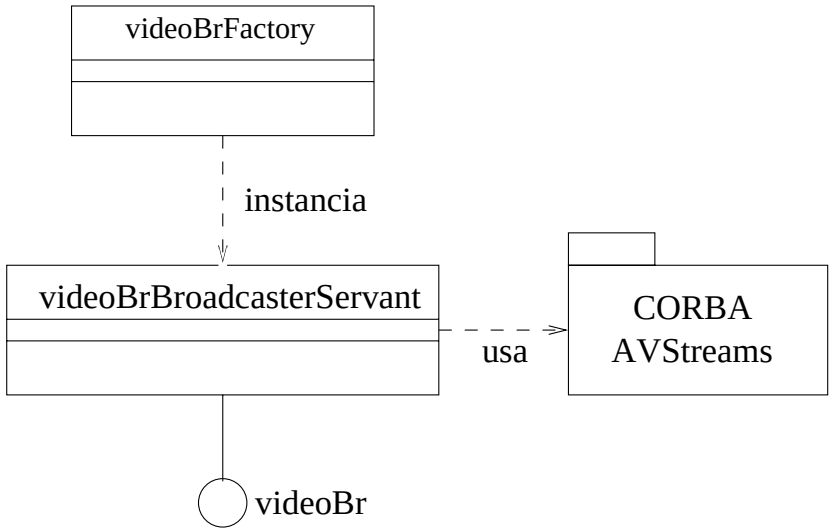

Figura 3.7: Arquitetura da interface broadcaster.

No segmento de código a seguir, pode-se observar um arquivo IDL gerado pela ferramenta ccmBuilder que possibilita operações na porta *broadcaster*.

```
interface videoBrBroadcaster : Video::F_video {
  Components::Cookie subscribe_videoBr(in Video::video_Consumer peer)
       raises(AVStreams_Full::alreadyConnected);

  void unsubscribe_videoBr(in Components::Cookie ck)
       raises(AVStreams_Full::notConnected);
  };
```

O método *subscribe_videoBr* permite que um consumidor de mídia subscreva-se para consumir a mídia produzida pela porta *broadcaster*. O método retorna um identificador de subscrição (*cookie*). O método *unsubscribe_videoBr* cancela a subscrição do consumidor utilizando o identificador fornecido pelo método *subscribe_videoBr*.

Porta *Player*

A porta *player* é uma interface consumidora de fluxo de mídia, que pode se conectar tanto a uma porta *transmitter* quanto a uma porta *broadcaster*. Um componente *player* é especificado em XML, como poder ser visto no fragmento de código a seguir.

```
<componentfeatures
   name  = "VideoPlayer"
   repid = "IDL:VideoPlayer:1:0">

  <ports>
    <!-- video player port -->
    <plays
       playsname = "videoPl"
       mediatype = "video">
    </plays>
  </ports>
</component>
```

A figura 3.8 ilustra o processo de criação da porta *Player*. A fábrica da porta (*VideoPlayerFactory*) instancia uma interface *FDev* do serviço *AVStreams*. Esta interface instancia outra interface (*FlowConsumer*) que é a porta consumidora de mídia que tanto pode ser áudio como vídeo. Ambas as interfaces pertencem ao serviço *A/VStreams*.

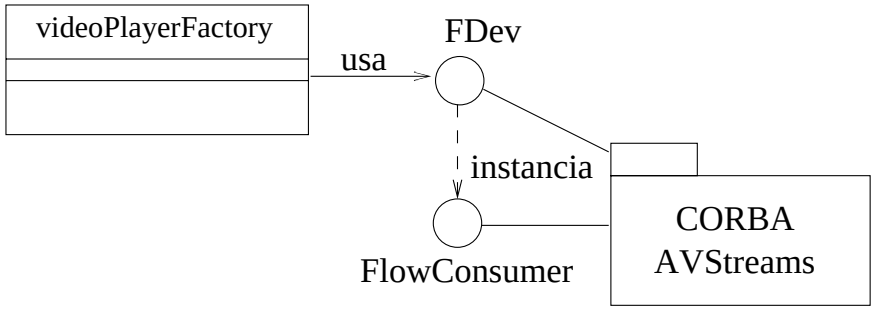

Figura 3.8: Arquitetura da interface player.

Porta *Emitter*

Porta *emitter* é responsável pela emissão de eventos de um tipo específico. A figura 3.9 exemplifica a arquitetura de uma porta emissora de eventos mostrando seu processo

de criação. A fábrica da porta (*URLSourceFactory*) instancia o *servant* (*URLSourceEmitterServant*) que implementa a porta e conseqüentemente disponibiliza a porta emissora de eventos (*URLString*). A figura ainda ilustra a possibilidade de interação entre a aplicação e a porta de emissora de eventos, onde um evento Java gerado pela aplicação é convertido em evento CORBA pela porta emissora para que o mesmo possa ser transmitido a uma porta consumidora conectada.

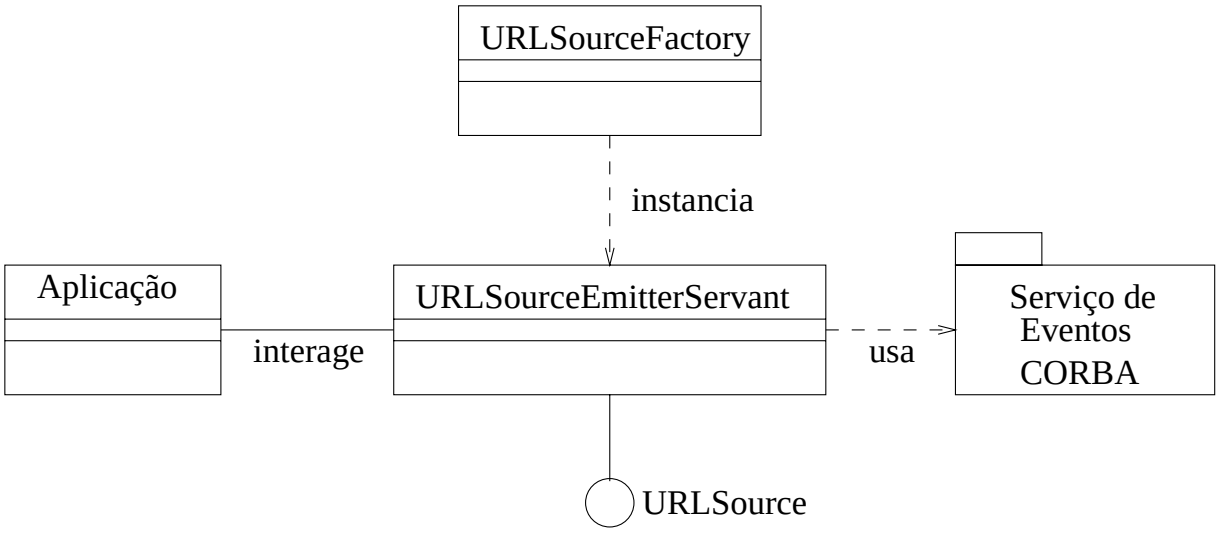

Figura 3.9: Arquitetura da interface emitter.

A porta emissora de eventos é especificada em XML através da marcação *emits*. O tipo de evento a ser emitido é especificado usando o atributo *eventtype* como se pode observar no código a seguir.

```
<ports>
  <emits
    emitsname = "URLSource"
    eventtype = "URLString">
    <eventpolicy policy = "normal"/>
  </emits>
</ports>
```

A ferramenta ccmBuilder gera uma interface IDL para operações na porta do componente. O fragmento de código a seguir ilustra dois métodos: *connect* para se conectar a porta emissora e um método *disconnect* que realiza a operação inversa.

```
// emitter  URLSource
interface URLSourceEmitter {
  void connect_URLSource
    (in CosEventComm::PushConsumer consumer)
```

```
  raises
    (Components::AlreadyConnected);

  void disconnect_URLSource()
  raises (Components::NoConnection);
};
```

Um segmento de código gerado pelo ccmBuilder que implementa o evento a ser emitido é ilustrado a seguir. Neste exemplo foi adicionado o código "*public string url*" que define o evento a ser utilizado.

```
valuetype URLString
 {
    // complete event definition here
    public string url;  //added by user

    // initializer
    factory init(in string _url);
 };
```

O desenvolvedor também terá de cuidar da implementação do evento, preenchendo a classes *URLStringImpl.java* que foi criada pela ferramenta ccmBuilder adicionando construtores para o evento em questão, como se pode observar no código a seguir.

```
import org.omg.CORBA.*;
public class URLStringImpl extends URLString {

  public URLStringImpl() {
  // complete event initialization here (default values)

  }

  // add here the factory with the proper initialization parameters

}
```

A classe que implementa a porta emissora disponibiliza um método que recebe um evento local Java converte para um evento CORBA e o envia para a porta consumidora conectada. Esta classe deve implementar o método com o seguinte protótipo:

```
  public void actionPerformed (ActionEvent ev)
```

Porta *Publisher*

A porta *publisher* é responsável pela emissão de eventos de um tipo específico, semelhante ao modelo *emitter*, porém com a possibilidade de originar eventos para muitos consumidores. A figura 3.10 ilustra a arquitetura necessária para instanciar a porta *publisher*. A fábrica da porta (*URLSourceFactory*) instancia o *servant* (*URLSourcePublisherServant*) que implementa a porta e conseqüentemente disponibiliza a porta publicadora de eventos (*URLSource*).

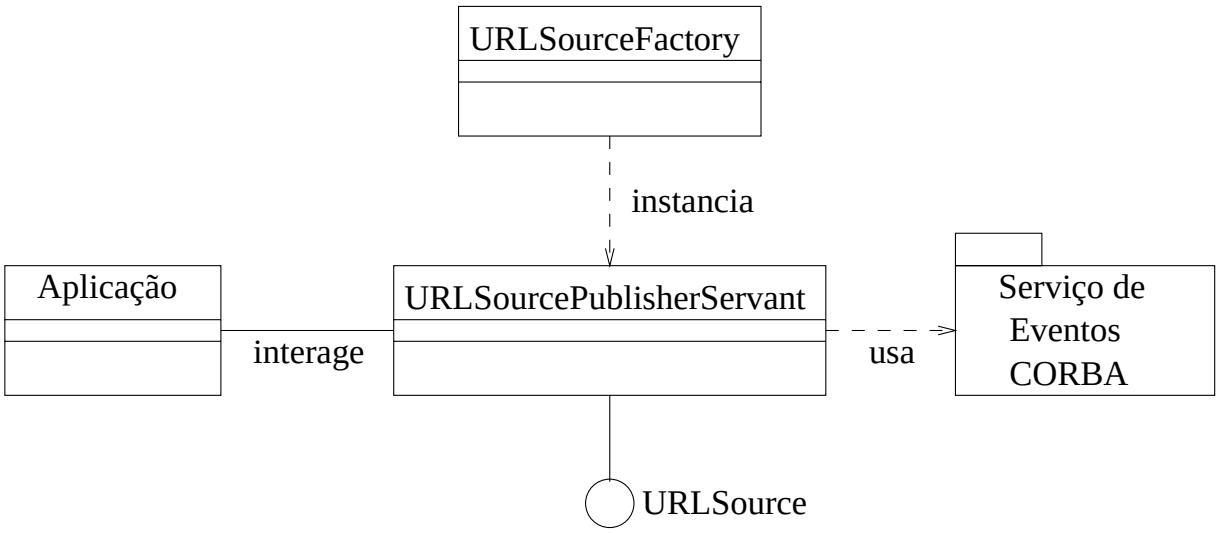

Figura 3.10: Arquitetura da interface publisher.

O nome da porta é especificado em XML usando-se a marcação *publishes*. O tipo de evento a ser publicado é especificado usando-se o atributo *eventtype* como se pode observar no fragmento de arquivo XML ilustrado a seguir.

```
<ports>
  <publishes
    publishesname = "URLSource"
    eventtype = "URLString">
    <eventpolicy policy = "normal"/>
  </publishes>
</ports>
```

A ferramenta ccmBuilder gera uma interface IDL para operações na porta do componente. O fragmento de código a seguir, ilustra dois métodos: *subscribe* para subscrever-se à porta publicadora e um método *unsubscribe* que realiza a operação inversa.

```
// publisher PU_URLSource
interface PU_URLSourcePublisher
   :CosEventComm::PushSupplier {
  Components::Cookie subscribe_PU_URLSource
```

```
    (in CosEventComm::PushConsumer consumer)
  raises
    (Components::ExceededConnectionLimit, Components::AlreadyConnected);

  void unsubscribe_PU_URLSource (in Components::Cookie ck)
    raises (Components::InvalidConnection);
};
```

A seguir pode-se observar um segmento de código gerado pelo ccmBuilder do evento a ser publicado. Este arquivo IDL é a representação de um evento chamado *URLString* que é um objeto por valor (*object by value*).

```
  valuetype URLString
 {
   // complete event definition here
   public string url;

   // initializer
   factory init(in string _url);
 };
```

Assim como na porta *emitter* o desenvolvedor terá de cuidar da implementação do evento, preenchendo a classe *URLStringImpl.java*.

Tal como portas emissoras, portas publicadoras de eventos definem o método actionPerformed para conversão e envio de eventos locais Java.

Porta *Consumer*

A porta *consumer* consome eventos de um tipo específico, podendo se conectar tanto à porta *emitter* quanto à porta *publisher*. A figura 3.11 ilustra a arquitetura necessária para instanciar a porta *consumer*. A fábrica da porta (*URLSinkFactory*) instancia o *servant* (*URLSinkConsumerServant*) que implementa a porta e conseqüentemente disponibiliza a porta consumidora de eventos (*URLSink*). A figura ainda ilustra a possibilidade de interação entre a aplicação e a porta de consumidora de evento, onde um evento CORBA enviado à porta é convertido em evento Java e transmitido para aplicação.

O nome da porta e o tipo de evento a ser consumido são especificados em XML conforme ilustrado a seguir.

```
<ports>
  <consumes
    publishesname = "URLSink"
    eventtype = "URLString">
    <eventpolicy policy = "normal"/>
  </consumes>
</ports>
```

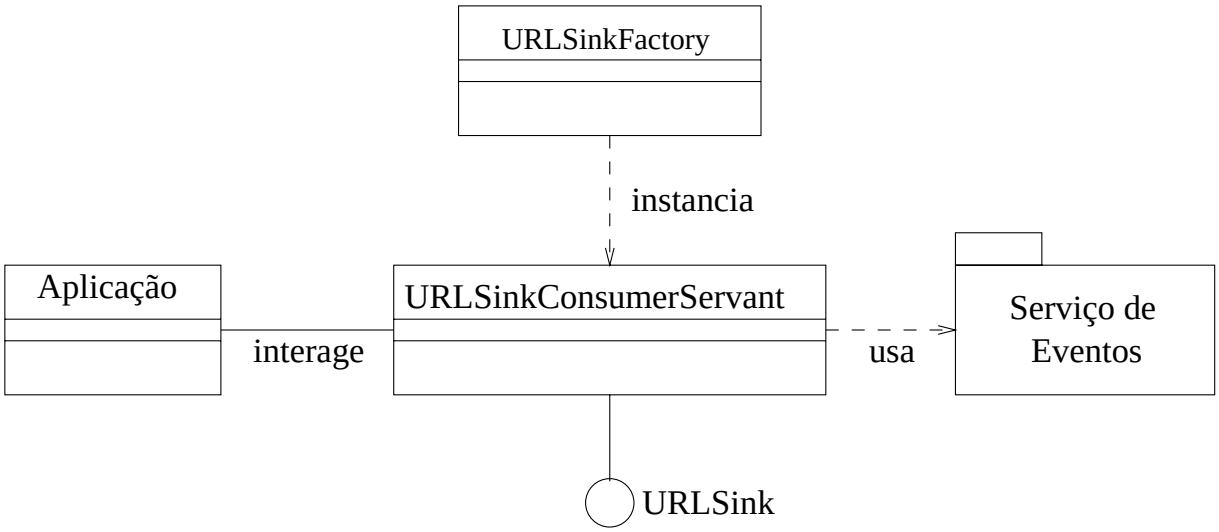

Figura 3.11: Arquitetura da interface consumer.

A seguir, pode-se observar um segmento de código gerado pelo ccmBuilder que implementa as operações definidas para a porta. O método *push* possibilita que eventos sejam empurrados para esta interface. O método *addActionListener* adiciona a referência de um objeto interessado em consumir eventos CORBA que internamente serão convertidos em eventos Java. O método *removeActionListener* retira a referência da lista de interessados em consumir eventos.

```
public void push(org.omg.CORBA.Any evt)
   throws CosEventComm.Disconnected {
   ...
}

void addActionListener(ActionListener l) {
  ...
}

void removeActionListener (ActionListener l) {
  ...
}
```

Porta *Facet*

Uma faceta é uma porta que provê métodos específicos da aplicação. A figura 3.12 ilustra a arquitetura de uma faceta.

A interface faceta é especificada em um documento XML. O exemplo a seguir ilustra uma especificação de uma faceta através da marcação *provides* onde o atributo *providesname* é responsável pelo nome da faceta.

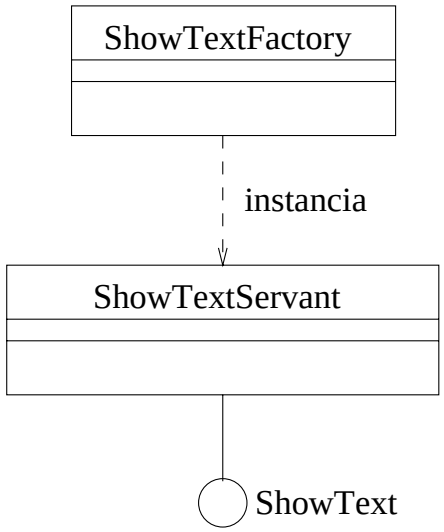

Figura 3.12: Arquitetura da uma faceta.

```
<ports>
  <!-- FACETS -->
  <provides
     providesname = "ShowText"
  </provides>
</ports>
```

Os métodos gerados pela ferramenta ccmBuilder devem ser supridos pelo desenvolvedor, posto que a lógica dos métodos depende da aplicação.

Interface *Receptacle*

Uma interface do tipo receptáculo é uma porta que provê conexões para o uso de referências. O receptáculo é realizado na interface equivalente. A figura 3.13 exemplifica a arquitetura de um receptáculo.

A interface receptáculo é especificada em um documento XML através da palavra reservada *uses*.

```
<componentfeatures
  name = "FullComp"
  repid = "IDL:FullComp:1.0">
<ports>
   <!-- RECEPTACLES -->
   <uses
      usesname = "receptacleR"
      multiple = "no">
   </uses>
</port>
```

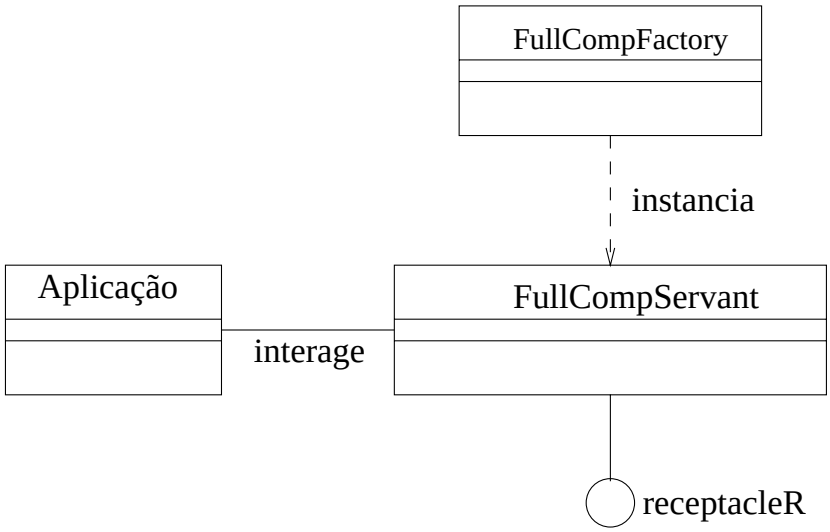

Figura 3.13: Arquitetura de um receptáculo.

```
</componentfeatures>
```

Os receptáculos podem ser simples ou múltiplos. Receptáculos simples permitem que somente uma referência seja conectada, enquanto receptáculos múltiplos permitem que várias referências sejam conectadas simultaneamente.

O código IDL a seguir ilustra um receptáculo simples.

```
// single receptacles
void connect_receptacleR(in Object facet)
  raises(Components::AlreadyConnected, Components::InvalidConnection);
Object disconnect_receptacleR() raises (Components::NoConnection);
Object get_connection_receptacleR();
```

O código IDL a seguir ilustra um receptáculo múltiplo.

```
// multiple receptacle
struct receptacleMConnection {
  Object objref;
  Components::Cookie ck;
};
typedef sequence <receptacleMConnection> receptacleMConnections;
Components::Cookie connect_receptacleM (in Object connection) raises
  (Components::ExceededConnectionLimit, Components::InvalidConnection);
Object disconnect_receptacleM (in Components::Cookie ck) raises
  (Components::InvalidConnection);
receptacleMConnections get_connections_receptacleM();
```

3.2.1 Exemplo de Componente

A figura 3.14 exemplifica um componente (*C_URLPresenter*) gerado para consumir eventos. O componente possui fábrica, a interface equivalente e a porta (*port*) consumidora de eventos com sua respectiva fábrica.

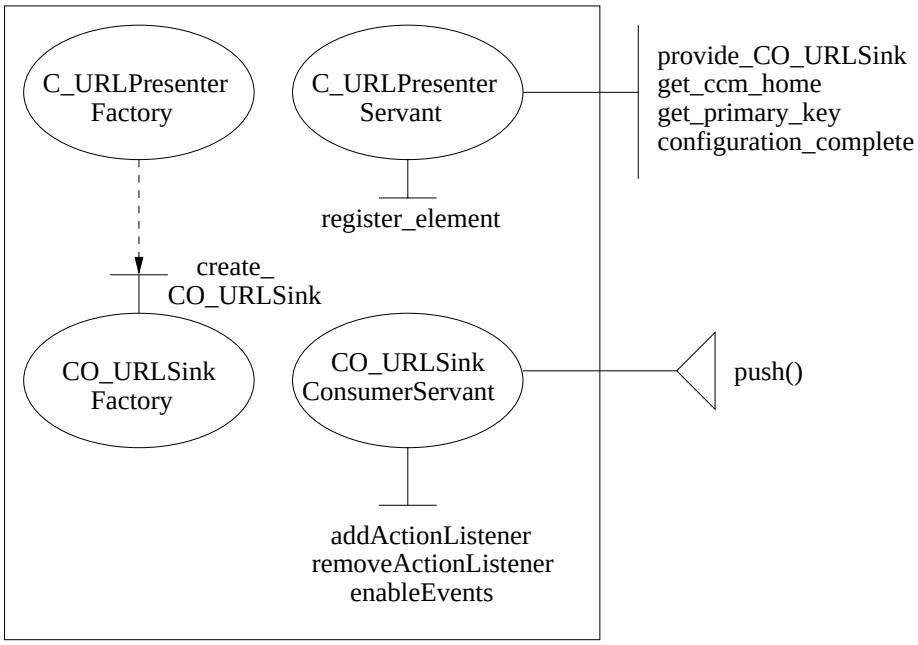

Figura 3.14: Classes criadas para um componente consumidor de eventos.

O componente é construído através da fábrica do componente. Esta fábrica irá instanciar a interface equivalente através da qual é possível a descoberta e uso da referência da porta consumidora de eventos. A interface equivalente provê os seguintes métodos: *provide_CO_URLSink*, *get_ccm_home*, *get_primary_key* e *configuration_complete*, onde o primeiro provê uma forma de se obter a referência da porta, o segundo método possibilita a obtenção da fábrica do componente, o terceiro devolve uma chave para acesso ao componente (chave primária) e o último método permite descobrir se o componente está pronto para o uso. O método *register_element*, interno ao componente, permite que uma porta se registre na interface equivalente.

CO_URLSinkFactory é uma fábrica. Esta fábrica irá instanciar a porta consumidora de eventos chamada *CO_URLSinkConsumerServant*. Esta porta é responsável por consumir eventos CORBA e transformá-los em eventos Java. Isto é possível através da operação *push*, onde eventos CORBA são empurrados nesta interface e então convertidos para eventos Java através dos métodos internos ao componente, para envio aos objetos que se subscreveram para recebê-los. Os métodos: *addActionListener* permite a adição à lista de subescritores aptos a consumir eventos Java, *removeActionListener* possibilita a remoção dos elementos interessados em consumir eventos e por último *enebleEvents* que habilita a recepção dos eventos CORBA.

3.3 *Container*

Containers são lugares onde os componentes são instanciados para execução. *Containers* fornecem os recursos necessários para um componente, por exemplo, recursos para processamento, comunicação e armazenamento. Assim como componentes, *containers* são especificados em XML. A especificação declara os tipos de componentes a serem instanciados pelo *container*, bem como suas respectivas dependências de outros componentes.

```
<container
   name = "VideoPanel"
   deployedas = "process">

  <!-- components that will be instantiated on this container -->
  <component
     name = "VideoConsumer">
     <dependson name = "VideoProducer"/>
  </component>

  <!-- POA -->
  <poapolicies
      thread = "ORB_CTRL_MODEL"
      lifespan  = "TRANSIENT"
      iduniqueness = "UNIQUE_ID"
      idassignment = "SYSTEM_ID"
      servantretention = "RETAIN"
      requestprocessing = "USE_ACTIVE_OBJECT_MAP_ONLY"
      implicitactivation = "IMPLICIT_ACTIVATION">
  </poapolicies>
</container>
```

Para o *container* serão criadas as seguintes classes e *interfaces* que poderão ser visualizadas na figura 3.15.

O processo de instanciação de um *container* consiste em criar as políticas do POA especificadas no descritor do *container* através da fábrica *POAFactory*. Entende-se por políticas do POA as regras que irão ditar como o servidor irá se comportar, como por exemplo, se o servidor será transiente ou persistente, se terá suporte a *multithread* ou não. O *container* disponibiliza a operação *find_home_by_name* que permite localizar um nome no *container*. Em seguida é gerada uma instância da classe *HomeFinderServant*. O *HomeFinderServant* é o único elemento que necessariamente necessita se registrar no serviço de nomes. Em seguida é criado um *home* para cada componente declarado (*C_URLPresenterHome*). Através do *home* é possível criar (*create*) ou remover (*remove*) um componente, obter uma chave primária para o componente (*get_primary_key*) e encontrar um componente utilizando determinada chave (*find_by_primary_key*).

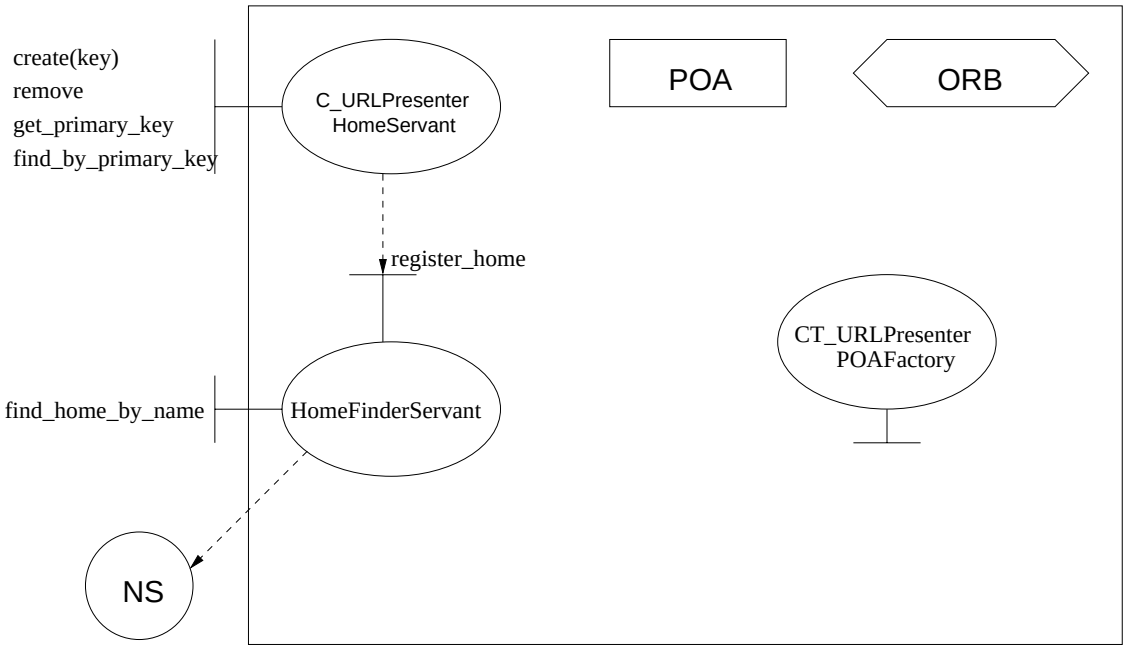

Figura 3.15: Classes e interfaces geradas para um *container*.

3.3.1 *Container* e o Componente

O *container* fornece toda a infra-estrutura necessária para que um componente possa existir. Na figura 3.16 podemos visualizar um *container* e um componente. A ferramenta gera, além do *container* e do componente, um arquivo de distribuição (*deployment*) que irá descrever as propriedades do ORB a ser utilizado, a URL (*Uniform Resource Locator*) do serviço de nomes, parâmetros de QoS (qualidade de serviço) e nome da aplicação. A ferramenta ccmBuilder também gera um arquivo para compilar os componentes gerados com a ferramenta Ant.

3.4 Distribuição (*Deployment*)

Containers são distribuídos (instalados) de acordo com as especificações descritas no descritor de distribuição (*deployment descritors*). Um descritor de distribuição declara os atributos necessários para configurar o *container* durante sua instalação. Um exemplo de descritor de distribuição especificado em XML é dado a seguir. Neste exemplo o descritor declara quatro parâmetros de qualidade de serviço para uma porta consumidora de fluxo e a forma como será instalado (processo ou applet Java).

```
<deployment
  class = "VideoPanel"
  deployedas = "applet"
  file = "faudio.html"
  archive = "props.jar,avs.jar,jmf.jar"
```

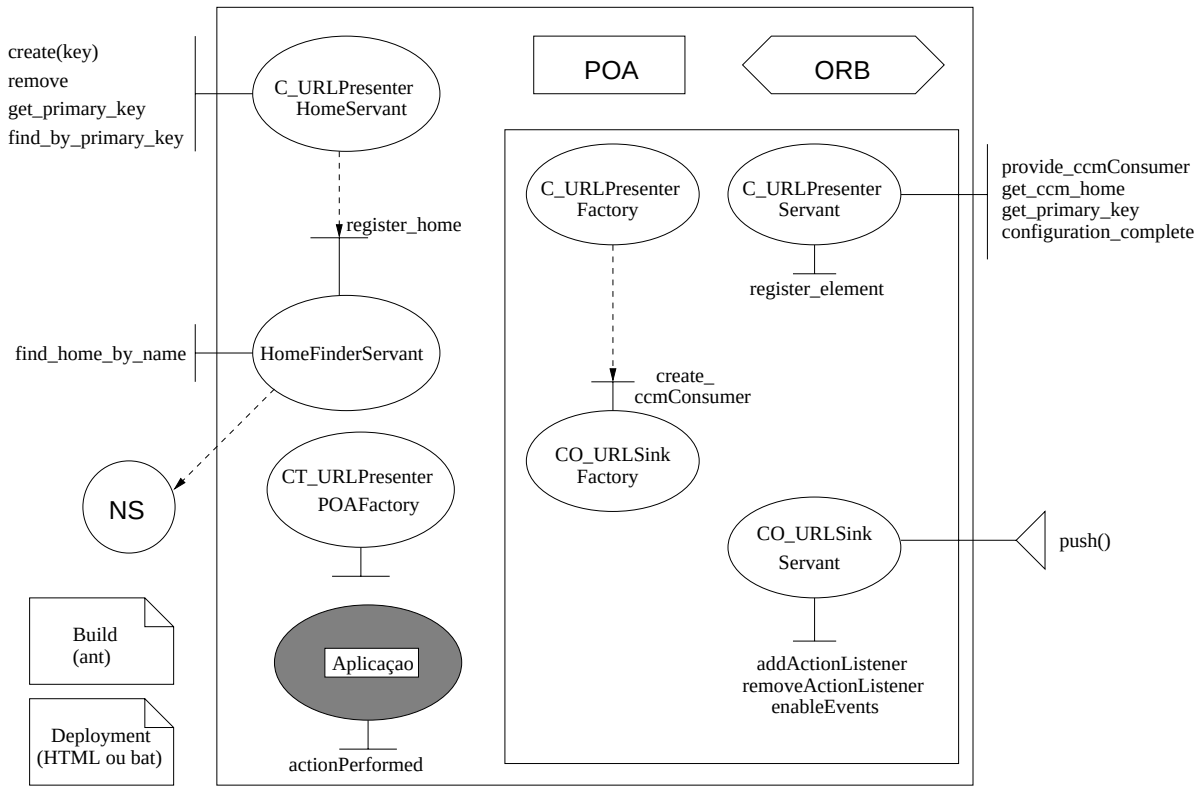

Figura 3.16: *Container* contendo um componente.

```
  codebase = "./"
  width = "320"
  height = "260">

  <ORB
    org.omg.CORBA.ORBClass = "com.sun.corba.se.internal.POA.POAORB"
    org.omg.CORBA.ORBSingletonClass = "com.sun.corba.se.internal.POA.POAORB">
  </ORB>

  <NameService
    NameServiceURL = "www.dca.fee.unicamp.br/~mig/Tnameserv.html">
  </NameService>

  <HomeFinder
    HomeFinderName = "VideoDisplay">
  </HomeFinder>

  <AVStreams>

     <VideoQoS
        video_framerate  = "15"
        video_colorDepth = "24"
        video_colorModel = "1"
        video_resolution = "160X120">
     </VideoQoS>

  </AVStreams>
</deployment>
```

No exemplo anteriormente ilustrado podemos observar as seguintes marcações e seus atributos:

- *class* - este atributo indica o nome da classe principal que será executada pela *applet* ou o arquivo a ser executado se for um processo;
- *deployedas* - indica se o componente será uma *applet* ou processo;
- *file* - nome do arquivo a ser gerado para distribuição. Se o atributo *deployedas* contiver o valor *applet* o arquivo gerado será no formato HTML. Se o atributo *deployedas* contiver o valor *process* o arquivo gerado será um arquivo de lote (*bat* ou *shell script*);
- *archive* - neste atributo deve-se inserir as bibliotecas (arquivos jar) necessários à aplicação;
- *codebase* - diretório onde se encontra o componente;

- *width* - largura da *applet*;
- *heigth* - altura da *applet*.

A marcação *NameService* possui o atributo *NameServiceURL* que permite obter a URL do serviço de nomes. A marcação *AVStreams* possui atributos que permitem configurar a forma como a mídia será apresentada.

Se o projeto para distribuição for do tipo processo, um arquivo do tipo *bat* será criado onde os valores obtidos do arquivo XML serão utilizados como argumentos na linha de comando que irá disparar a aplicação. Caso seja do tipo *applet*, ccmBuilder irá gerar um arquivo HTML e os valores lidos do arquivo XML serão incorporados no arquivo HTML como parâmetros, como pode ser visto no código a seguir.

```
<HTML>
<BODY>
<APPLET CODE = "VideoPanelContainerApp.class"
  CODEBASE = "./"
  ARCHIVE = "props.jar,avs.jar,jmf.jar"
  WIDTH = "320" HEIGHT = "260">

  <PARAM NAME = "org.omg.CORBA.ORBClass"
    VALUE = "com.sun.corba.se.internal.POA.POAORB">
  <PARAM NAME = "org.omg.CORBA.ORBSingletonClass"
    VALUE = "com.sun.corba.se.internal.POA.POAORB">
  <PARAM NAME = "NameServiceURL"
    VALUE = "www.dca.fee.unicamp.br/~mig/Tnameserv.html">
  <PARAM NAME = "HomeFinderName"  VALUE = "VideoDisplay">
  <PARAM NAME = "video_framerate"  VALUE = "15">
  <PARAM NAME = "video_colorDepth"  VALUE = "24">
  <PARAM NAME = "video_colorModel"  VALUE = "1">
  <PARAM NAME = "video_resolution"  VALUE = "160X120">
</APPLET>
</BODY>
</HTML>
```

3.5 Resumo do Capítulo

Neste capítulo foi descrita a arquitetura de suporte do modelo de componentes CM-Tel, seus elementos funcionais e a estratégia de geração de código para este modelo através da ferramenta ccmBuilder. No capítulo seguinte iremos abordar as decisões de projeto para o modelo descrito neste capítulo e os detalhes de implementação.

Capítulo 4

Implementação da Ferramenta ccmBuilder

Este capítulo descreve as decisões de projeto para o modelo descrito no capítulo anterior e os detalhes de implementação, como o tipo de parser XML escolhido, as principais estruturas internas da implementação, os serviços CORBA empregados (eventos, propriedades, *streams*) e as demais tecnologias empregadas para a construção da ferramenta ccmBuilder (Java, CORBA, análise léxica).

4.1 Representação de um Componente em XML

Dado o impacto que teria a adoção de extensões à IDL, o modelo CM-Tel optou não pela adição de novas sintaxes na IDL, como é proposto no CCM, mas pela representação do componente em XML. Entretanto, surge agora um novo problema: como representar o componente em XML. Para isso, baseamo-nos na especificação do CCM, que possui uma representação dos componentes em XML no modelo de distribuição. Portanto, precisaríamos adaptar parte do modelo de distribuição para o modelo abstrato.

XML Schema

XML*Schema*[20] fornece uma linguagem para descrever tipos em XML. A linguagem é expressa em XML e inclui facilidades para definir tipos estruturados e textuais, incluindo tipos derivados de outros tipos. Tipos estruturados são usados para descrever elementos que possuem elementos filhos ou atributos associados a ele. Tipos textuais são usados para elementos que contenham somente texto e para valores dos atributos. A versão atual da ferramenta ccmBuilder utiliza XML *Schema*.

Parser XML

Um *parser* XML[8][9] é um componente de *software* que processa arquivos XML extraindo seus elementos e armazenando-os em uma estrutura de dados adequada à navegação através destes elementos. Sua tarefa é esconder do desenvolvedor a complexidade

da sintaxe da linguagem XML. Documentos XML podem ser tanto bem-formados quanto válidos. Documentos bem formados respeitam as regras sintáticas. Documentos válidos não somente respeitam as regras sintáticas, como também devem estar em concordância com uma estrutura descrita por um DTD. A finalidade do *parser* consiste em ler e interpretar um documento XML para descobrir as entidades neles armazenadas e verificar a validade do documento XML confrontando-os com um DTD.

O *parser* XML e a aplicação compartilham uma estrutura de árvore na memória, como pode ser visto na figura 4.1. O *parser* lê os documentos XML e preenche a árvore na memória. A aplicação manipula estes dados como se estivesse manipulando o documento XML.

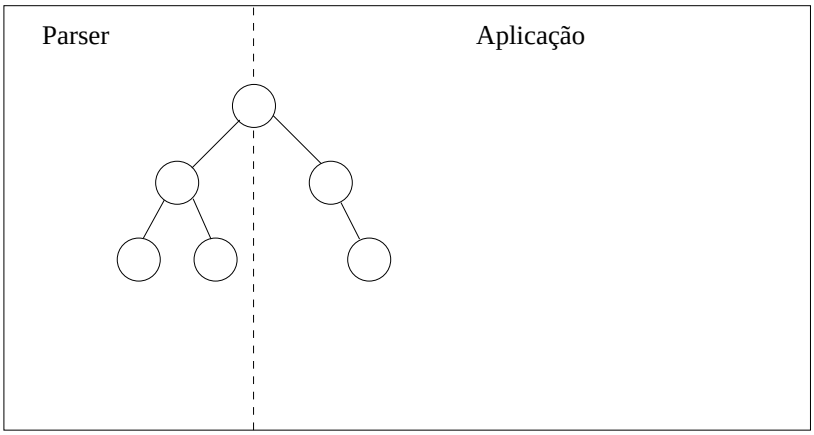

Figura 4.1: Arquitetura de um programa XML.

4.1.1 O Parser xml4j

A IBM fornece um *parser* chamado xml4j[21], que foi totalmente escrito em Java. Este *parser* utiliza DTD para validar documentos XML e possui suporte tanto para o modelo DOM (*Document Object Model*)[22][23] quanto ao modelo SAX (*Simple API for XML*)[20]. O modelo DOM representa o documento XML em um formato hierárquico extraindo seus elementos e armazenando-os em uma estrutura de dados adequada à navegação através destes elementos. O modelo SAX é um conjunto de interfaces abstratas que analisa um documento XML em uma cadeia de chamadas a métodos bem formados. SAX fornece um modelo de encadeamento que pode ser usando tanto para processar documentos XML quanto para produzir documentos XML. A seguir, iremos detalhar alguns métodos contidos na API do xml4j e utilizadas no ccmBuilder.

startElement

Este método é chamado sempre que uma marcação for encontrada. Através dos atributos *StringTagName* e *attributeList* podemos recuperar as informações armazenadas no documento XML. O atributo *StringTagName* retorna o nome da marcação e *attributeList* retorna uma lista que é composta de nome e valor.

```
public static void startElement
  (String stringTagName, AttributeList attributelist){
...
}
```

endElement

Este método é chamado sempre que o *parser* sair de uma marcação. O atributo *String-TagName* retorna o nome da marcação que o *parser* acabou de processar.

```
public static void endElement (String stringTagName)
{
...
}
```

characters

Este método permite recuperar os valores textuais de um documento XML.

```
public static void
  characters(char [] rgc, int nStart, int nLength)
{
  String x = new String(rgc,nStart,nLength);
...
}
```

STParser

STParser é um analisador léxico simplificado implementado em Java. O STParser é uma especialização da classe Java *StreamTokenizer*. *StreamTokenizer* é uma classe que processa um fluxo de caracteres de entrada quebrando o fluxo em *tokens* e permitindo assim que os *tokens* possam ser lidos um a um. Utilizamos como delimitador de *tokens* o caractere $. Quando um *token* específico é identificado, ele é substituído pelo valor obtido na árvore que foi previamente preenchida pelo *parser* XML. Resumidamente, a finalidade do STParser é obter os elementos da árvore em memória e preencher os *templates* gerando como arquivos de saída os componentes, *containers*, descritores de distribuição e arquivos do tipo *make* em XML para serem utilizados a posteriori pela ferramenta Ant[15][16].

A figura 4.2 ilustra as operações realizadas pelo ccmBuilder que consistem em obter os elementos e atributos de um arquivo XML e construir uma tabela de substituição. Esta tabela de substituição será utilizada para transformar os *templates* em códigos. Isto ocorre trocando as marcações contidas nos *templates* pelo valor obtido no arquivo XML. Podemos observar melhor o processo se analisarmos o exemplo a seguir, que é ilustrado na Figura 4.2. O ccmBuilder irá ler o arquivo XML e obter a marcação <*transmits*> que indicará quais *templates* serão utilizados e como será montada a tabela de substituição. Na figura pode-se observar que tabela de substituição será composta, para este exemplo, dos seguintes itens:

- *+++transmitsname+++* : esta marcação irá nomear a porta do componente, e neste exemplo, o nome do componente será VideoTr.
- *+++mediatype+++* : indica o tipo de mídia a ser utilizada, que tanto pode ser vídeo ou áudio. Para esta marcação, o valor de substituição sempre será em minúsculo (video).
- *+++MediaType+++* : semelhante à marcação +++mediatype+++, porém o valor de substituição será composto de maiúscula e minúsculas (Video).

STParser, neste exemplo, troca os *tokens $+++transmitsname+++$, $+++mediatype+++$* e *$+++MediaType+++$* pelos valores obtidos na tabela de substituição VideoTr, video e Video respectivamente, formando os arquivos de saída, como se pode observar na figura 4.2.

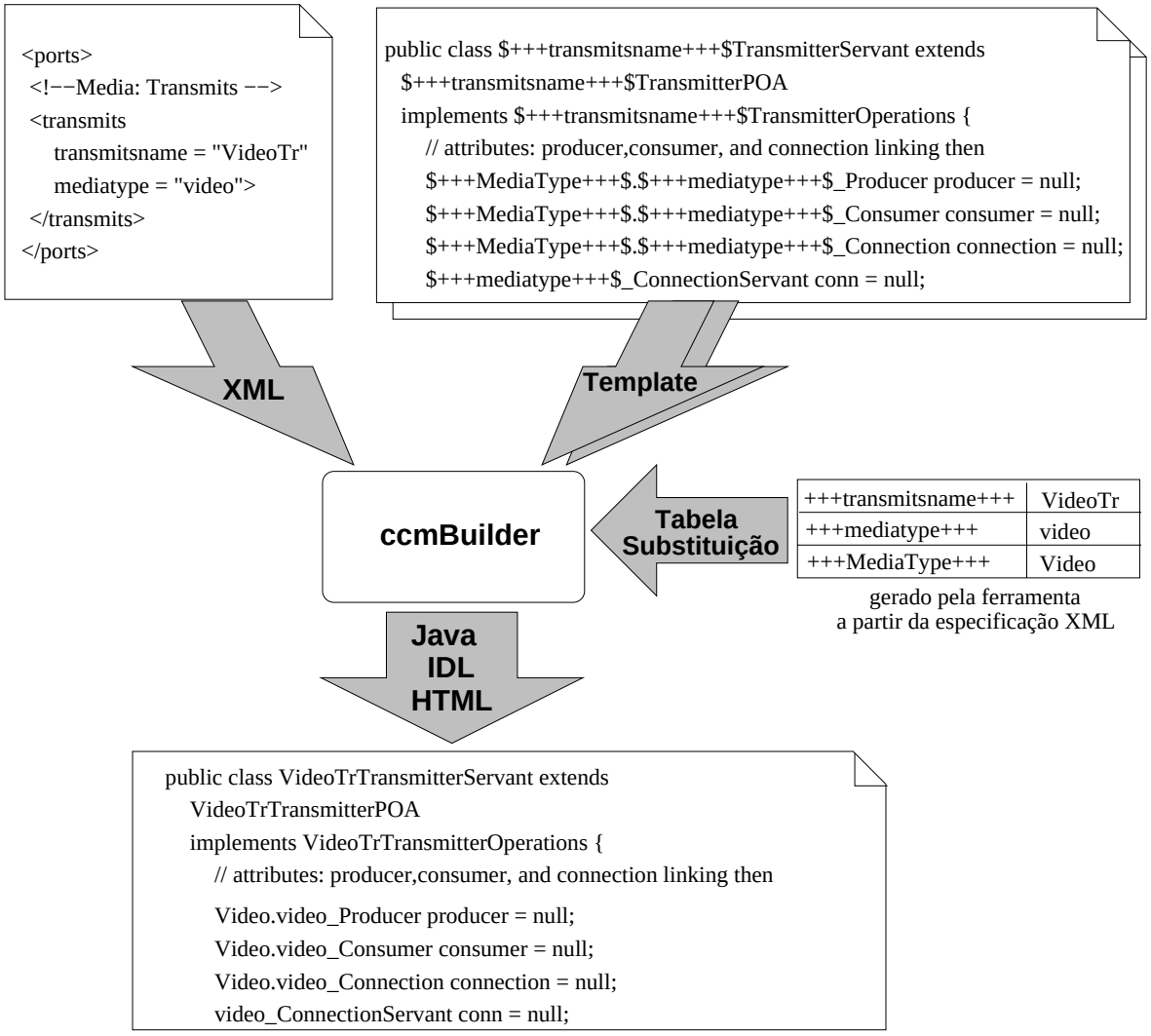

Figura 4.2: Criação de arquivo Java/IDL a partir de um arquivo XML.

4.2 Estratégia de Geração de Código

O processo de geração de código possui o ciclo ilustrado na Figura 4.3. Inicialmente, o projetista modela o software usando ferramentas tradicionais encontradas no mercado como, por exemplo, a ferramenta *ROSE* da *Rational*. O modelo do projeto é então manualmente transposto para arquivos XML contendo a especificação do componente. Estes arquivos são submetidos ao ccmBuilder que irá gerar as interfaces das portas (IDLs) e a implementação dos *Homes*, portas e fábricas (Java). As interfaces das portas geradas pelo ccmBuilder serão preenchidas pelo desenvolvedor com os métodos das facetas e com as definições dos objetos por valor (*valuetypes*). Devidamente preenchidas, as interfaces são submetidas a um compilador IDL (idlj) que irá gerar os *stubs e skeletons* das interfaces. Finalmente, um compilador Java transforma os elementos acima gerados em um *framework* de componentes. Para facilitar o processo de compilação, foi utilizado a ferramenta Ant do projeto Jakarta[15][16].

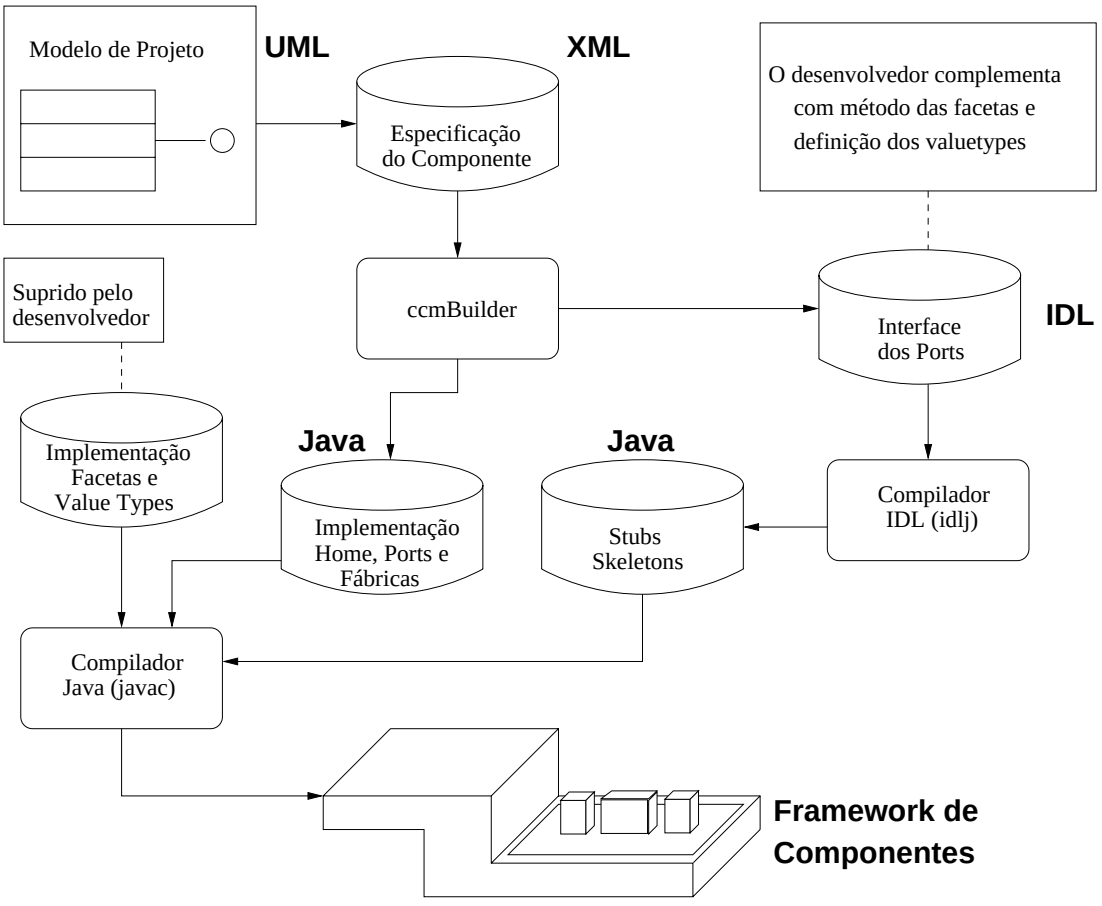

Figura 4.3: Processo de geração de código para a arquitetura proposta.

A ferramenta pode ser decomposta em três grandes classes como ilustra a Figura 4.4. A primeira classe, ccmBuilder permite fazer a escolha do arquivo a ser processado. A segunda classe, ccmParser, possui funcionalidades que viabiliza o uso dos elementos contidos em

um arquivo XML. A terceira classes STParser, permite a manipulação dos *templates* dos arquivos.

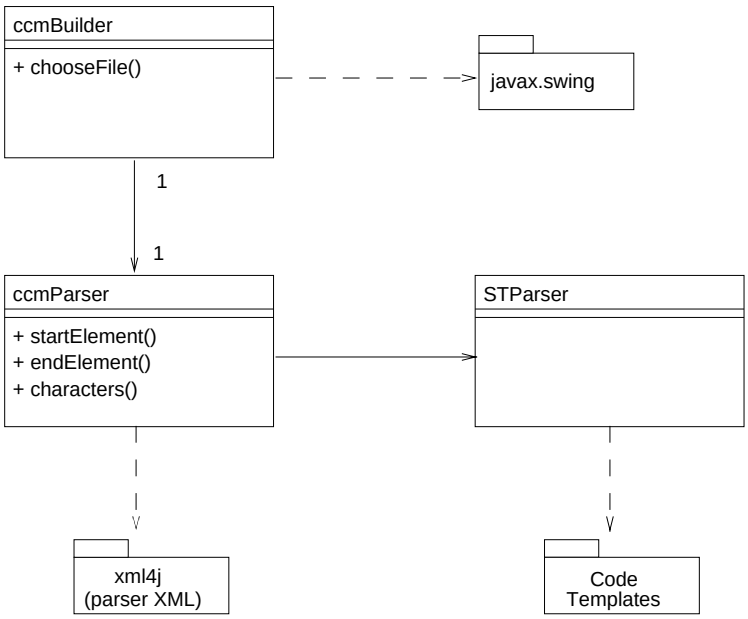

Figura 4.4: Diagrama de classes do ccmBuilder.

Na figura 4.5, onde um diagrama de seqüência ilustraa operação da ferramenta em função do tempo. A classe ccmBuilder, permite ao usuário navegar entre os diretórios e escolher o arquivo XML a ser processado pela ferramenta. Esta operação de navegar e selecionar o arquivo desejado é realizada pelo objeto *filechooser*, que é uma especialização da classe *FileChooser* contida no pacote *javax.swing*. Esta interface fornece uma forma amigável de escolher a especificação do componente que foi representada em um arquivo XML. Uma vez obtido o arquivo XML, o mesmo é transmitido à segunda classe, ccmParser, que estende as funcionalidade do *parser* XML para Java (xml4j)[24]. O *parser* irá obter os valores armazenados nas marcações do arquivo XML e submetê-los à terceira classe, STParser. Esta classe, STParser, irá fazer as substituições das marcas (*tokens*) contidas nos *templates* pelos valores fornecidos pelo ccmParser.

4.3 CORBA e Adaptadores de Objetos

Um adaptador de objetos [25] é um mecanismo que conecta uma requisição usando uma referência de um objeto com o código para o serviço que está sendo requisitado. Um adaptador de objetos define como um objeto é ativado, criando um novo processo, criando um *thread* para um processo existente ou reusando um processo ou *thread* existente.

Com a intenção de evitar a proliferação de tipos diferentes de adaptadores de objetos, o OMG no padrão CORBA 2.0 especificou um Adaptador de Objetos Básico (*Basic Object Adapter* - BOA).

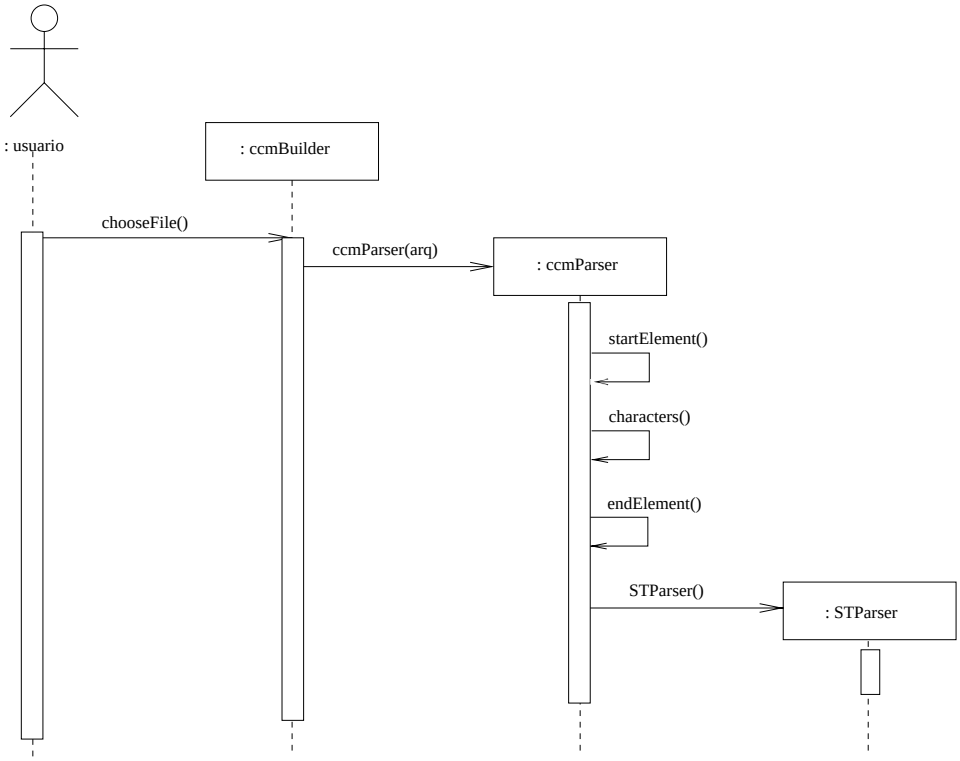

Figura 4.5: Diagrama de seqüência do ccmBuilder.

CORBA 2.0 torna clara a divisão entre servidor e objeto. Um servidor é uma unidade de execução (processo), enquanto um objeto implementa uma interface.

BOA torna o lado do cliente bastante portátil porém o lado do servidor não é portátil. O problema é que BOA não foi especificado por completo; conseqüentemente, os vendedores de ORB introduziram suas próprias funcionalidade ao BOA para preencher os vazios existentes na sua especificação.

Para não criar um problema maior que o existente, não foi criada nenhuma correção ao BOA. O que se escolheu foi criar uma nova versão desde o início. Esta nova versão, chamada Adaptador de Objetos Portátil (*Portable Object Adapter* - POA), pode ser considerada como o BOA especificado da forma correta. Assim como no BOA, é possível:

- lidar com ativação de objeto de forma transparente;
- iniciar um servidor para cada método;
- iniciar um programa separado para cada método;
- iniciar um programa compartilhado para todas as instâncias de um tipo de objeto.

POA suporta objetos transientes ou persistentes. Objetos transientes "vivem" somente enquanto o processo existir. Objetos persistentes "vivem" além da existência de um processo. Seu estado pode ser salvo e mais tarde recuperado.

POA também introduz um novo conceito, onde opcionalmente pode ser suprido um *servant manager* para cada implementação de uma interface de um objeto. Eles são objetos de *callback* que criam *servants* por demanda. O POA invoca operações no gerente de *servant* para criar, ativar e desativar *servants*. Pode-se não implementar o gerente de *servant* e deixar que o POA implicitamente ative e desative os *servants*.

Até então, para usufruir as propriedades do POA era necessário usar soluções proprietárias ou ferramentas que estavam surgindo, em seu formato embrionário na maiorias das vezes. Ferramentas como JacORB [26], OpenORB [27] e outras surgiram no decorrer das pesquisas como opções para viabilizar a implementação do CCM. Porém, estas ferramentas exigiam configurações adicionais nem sempre documentadas o suficiente, gerando com isso uma sobrecarga na implementação da ferramenta ccmBuilder. Foram feitos testes com a ferramenta JacORB e os resultados obtidos foram satisfatórios. Contudo, a adoção do POA no Java 1.4 de forma nativa facilitou a codificação e projeto do ccmBuilder. Por este motivo, foi escolhido o Java 1.4 pois tanto CORBA como POA já estão embutidos no pacote de distribuição da SUN, facilitando o desenvolvimento e distribuição dos componentes gerados pela ferramenta ccmBuilder.

4.4 Serviços CORBA

A seguir serão detalhados os serviços CORBA utilizados neste trabalho. Os serviços foram implementados em Java por outros membros do grupo para suportar o modelo CM-Tel.

4.4.1 Serviço A/V Streams

A Especificação *A/V Streams* [28] foi definida pela OMG para facilitar o desenvolvimento de aplicações multimídia distribuídas. *A/V Streams* é uma infraestrutura baseada em CORBA e composta de um conjunto de objetos distribuídos, com interfaces padronizadas, que implementam um *framework* de *streams* multimídia. Este *framework* possui suporte para conexões "ponto-a-ponto" (*unicast*) e "ponto-multiponto" (*multicast*), suporte a múltiplos protocolos (TCP, UDP, RTP/UDP e ATM/AAL5).

A especificação *A/V Streams* tem por objetivo permitir a interoperabilidade entre aplicações multimídia distribuídas de diferentes fornecedores, bem como entre aplicações e outros produtos baseados em CORBA.

4.4.2 Serviço de Eventos

O Serviço de Eventos [25] torna possível que objetos registrem/cancelem seu interesse em um evento específico, dinamicamente.

Normalmente, os objetos que geram os eventos não sabem quais objetos estão interessados em consumir eventos. Quem gerencia a interconexão entre produtores e consumidores é o Serviço de Eventos, que cria um canal de comunicação entre estes objetos.

O Serviço de Eventos define dois tipos de objetos: produtores (*suppliers*) e consumidores (*consumers*). Os produtores produzem eventos, enquanto os consumidores processam os eventos através de um manuseador de eventos. Os eventos são comunicados entre produtores e consumidores usando-se uma requisição CORBA.

Existem dois modelos para eventos: *push* e *pull*. No modelo *push*, o produtor de eventos toma a iniciativa de enviar os eventos para o consumidor. No modelo *pull*, o consumidor é quem toma a iniciativa e solicita eventos para o produtor.

O canal de eventos suporta ambos os modelos de notificação "*push* e *pull*", como são mostrados na Figura 4.6.

O Canal de Eventos

O canal de eventos genérico não entende o conteúdo dos dados que estão passando por ele. Os produtores e consumidores definem uma semântica de eventos comum entre eles. Contudo, o serviço de eventos também suporta o modelo de eventos tipados, que permite a aplicação descrever o evento usando IDL. Eventos tipados suportam tanto o modelo *push* quanto o modelo *pull*.

4.4.3 Serviço de Propriedades

Propriedades são valores nomeados e tipados associados dinamicamente a um objeto, fora do sistema convencional de tipos.

O Serviço de Propriedades [29][30] implementa objetos que suportam a interface *PropertySet* ou a interface *PropertySetDef*. A interface *PropertySet* suporta um conjunto de propriedades. Uma propriedade é um conjunto de pares: <*property_name,property_value*>, onde:

- *property_name* é uma string com o nome da propriedade;
- *property_value* é do tipo *any* e armazena o valor associado à propriedade.

A interface *PropertySetDef* é uma especialização da interface *PropertySet*.

O diagrama de classes ilustrado na Figura 4.7 representa as classes que compõem o Serviço de Propriedades.

Modelo de Propriedade do Cliente

Com os atributos CORBA, o cliente pode obter ou alterar os valores da propriedade. Clientes podem manipular propriedades individualmente ou em grupos usando uma seqüência de propriedades chamada *Properties*.

Quando usamos objetos que suportam a interface *PropertySetDef*, os clientes podem criar e manipular propriedades e suas características. A interface *PropertySetDef* também fornece operações ao cliente para recuperar informações de restrições de um *PropertySet*, como os tipos das propriedades que estão habilitados.

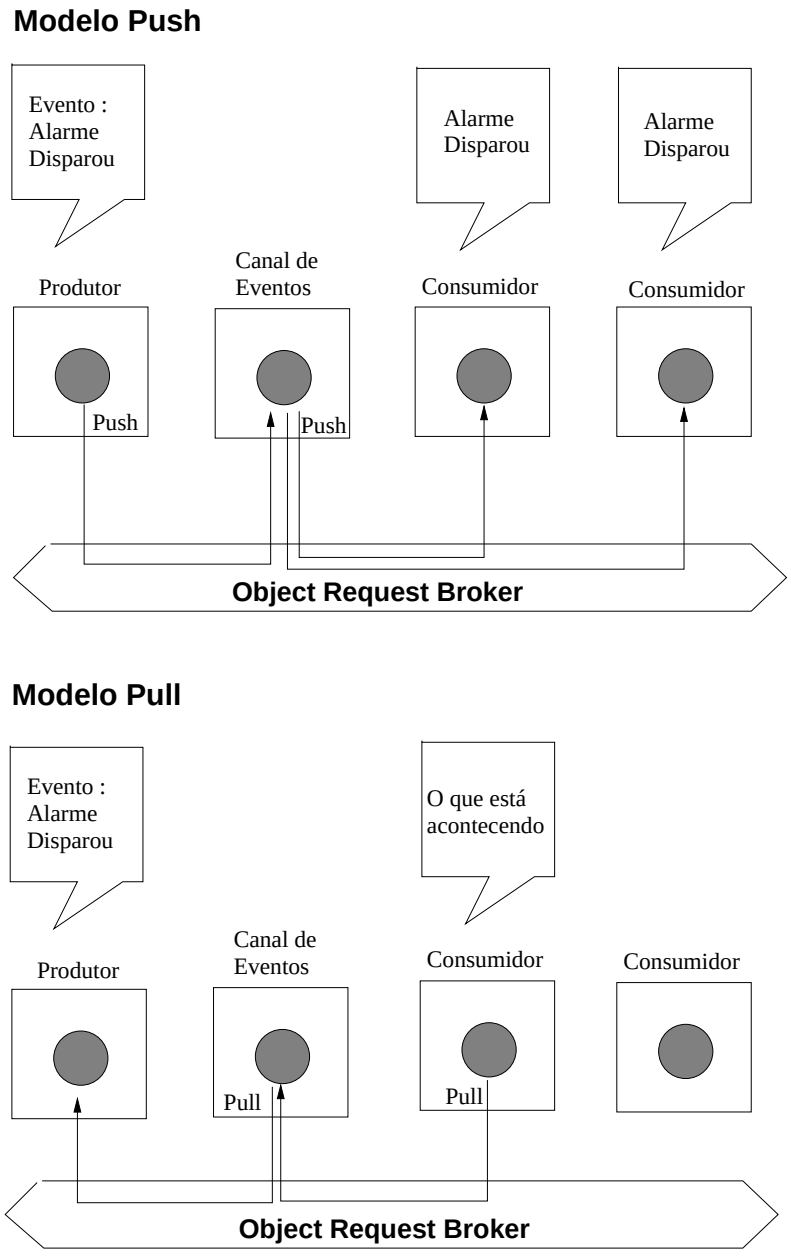

Figura 4.6: Serviço de eventos: modelo push e pull.

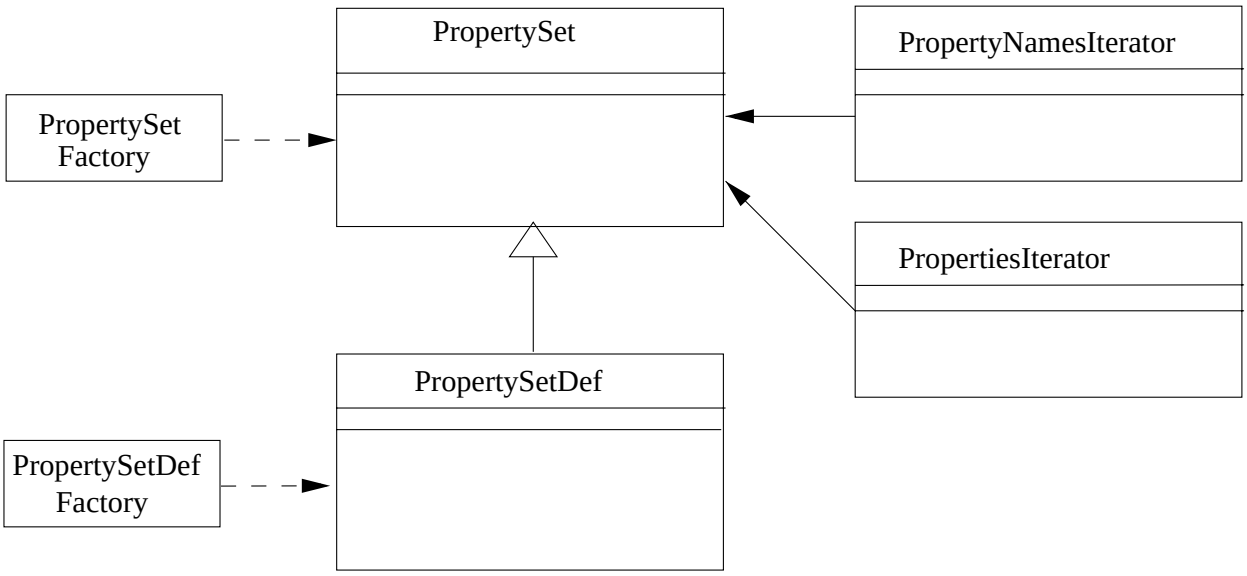

Figura 4.7: UML para o serviço de propriedades.

O cliente pode solicitar uma lista de nomes das propriedades (*PropertyName*) ou a quantidade de propriedades. *Iterators* são usados pelo serviço de propriedades para retornar uma lista de propriedades.

Modelo de Propriedade do Objeto

Objetos que desejam suportar o serviço de propriedades devem suportar a interface *PropertySet* ou *PropertySetDef*. *PropertySet* é uma interface que fornece operações para definir, remover, enumerar ou verificar a existência de propriedades. A interface *PropertySetDef* é uma subclasse de *PropertySet* e provê operações para recuperar restrições do *PropertySet*, definir e modificar os modos da propriedade.

A interface *PropertySet* provê exceções para viabilizar o conceito de propriedade que somente pode ser lida (*readonly*) e fixa (*fixed*). Propriedades fixas são aquelas que não podem ser removidas.

Quando um objeto *PropertySet* recebe uma requisição *define_property* do cliente, o objeto deve garantir que não existem conflitos no nome da propriedade e então reter a informação da propriedade para que possa responder, mais tarde, com invocações dos métodos *get_property, delete_property* e *is_property_defined*.

4.5 A Ferramenta Ant

A ferramenta *Apache Ant*[15][16] do *The Jakarta Project* é uma ferramenta de construção baseada em Java e pode ser considerada uma ferramenta do tipo *Make* para Java. *Ant* foi utilizada para compilar os arquivos produzidos pela ferramenta de geração de código.

A vantagem do uso do *Ant* reside na sua portabilidade. Ao contrário dos modelos onde se usa comandos baseados em *shell script*, *Ant* usa classes Java. Os arquivos de configuração são escritos usando um arquivo XML.

4.6 Resumo do Capítulo

Neste capítulo foram abordadas as tecnologias utilizadas pela ferramenta de geração automática de código (ccmBuilder) como, XML, DTD, parser utilizado (xml4j), CORBA e serviços CORBA (eventos, propriedades e fluxo de mídia). No capítulo seguinte, iremos mostrar um estudo de caso onde um foi construído um exemplo utilizando a ferramenta ccmBuilder.

Capítulo 5

Utilização da Ferramenta ccmBuilder

Neste capítulo será apresentado um protótipo, ccmVF, uma aplicação de vídeo-conferência desenvolvida para validar a ferramenta ccmBuilder.

Também serão discutidos os passos necessários para a construção do *software* de vídeo-conferência ccmVF. Finalmente, serão demonstradas as vantagens da construção de aplicações utilizando-se o ccmBuilder.

5.1 Suporte para a Implementação do ccmVF

O objetivo principal para o desenvolvimento do protótipo foi avaliar a adequação da ferramenta de geração de código (ccmBuilder). Por adequação, entende-se características como: simplicidade, abrangência, modularidade, flexibilidade e composição.

O protótipo é uma ferramenta de vídeo-conferência, cuja interface é apresentada na figura 5.1 e foi implementado como uma Applet Java compilado na versão 1.4.0 ou superior.

Os serviços CORBA utilizados pelo protótipo (propriedade, eventos e fluxo de mídia) foram escritos em Java, empacotados em um arquivo JAR e assinados para que pudessem ser descarregados através do protocolo HTTP. O serviço de fluxo de mídia emprega para captura e apresentação de áudio e vídeo a API *Java Media Framework* (JMF).

Os testes foram feitos em máquinas PC Pentium III 1 GHz e sistema operacional Windows 2000, uma vez que não foi possível a captura de áudio/vídeo em máquinas Linux com o JMF. A apresentação de áudio/vídeo no Linux foi satisfatória. Um servidor HTTP sob a plataforma UNIX armazenou os componentes e *containers* em arquivos JAR prontos para serem utilizados pelos usuários via *download* e a página HTML para acesso ao protótipo.

5.2 Exemplo de Uso do ccmVF

Para se estabelecer uma conexão, primeiramente, cada participante deve informar seu nome de usuário, seu endereço eletrônico e o endereço eletrônico com que deseja estabelecer uma conexão. Estas informações são utilizadas para registro e/ou consultas no serviço de nomes.

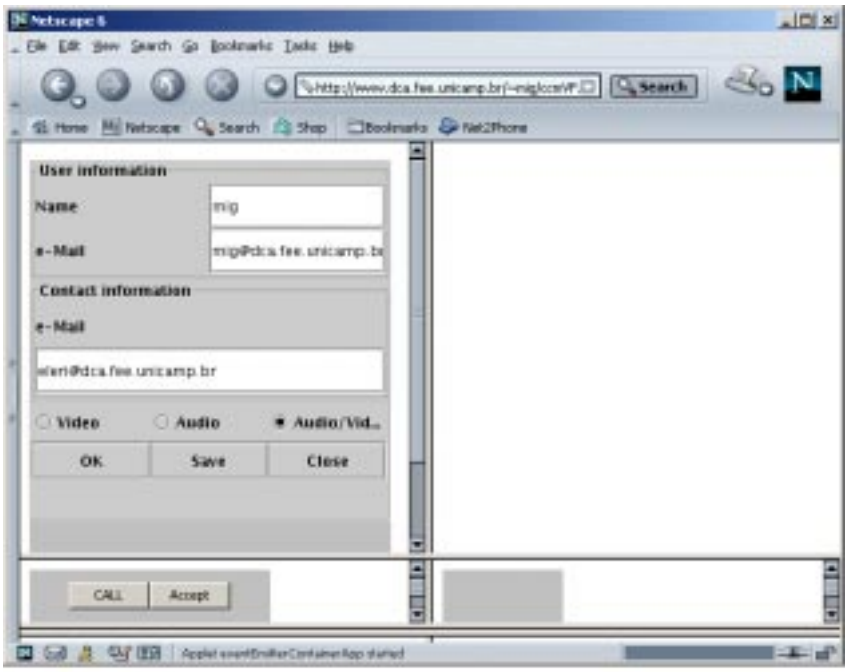

Figura 5.1: Tela do ccmVF.

Devidamente preenchidos os dados iniciais, o usuário deve informar os recursos a serem utilizados, por exemplo, áudio, vídeo ou áudio e vídeo simultaneamente. Esta opção foi agregada ao protótipo, tendo em vista que nem todos teriam o poder computacional necessário para transmissão de áudio e vídeo simultaneamente.

Em seguida, o participante deve pressionar o botão OK para que seja iniciado o componente de eventos. O componente de eventos será utilizado para solicitar ao outro participante da vídeo-conferência o estabelecimento de uma conexão. O componente de eventos irá incluir uma página com os seguintes botões: *CALL e Accept* (figura 5.2). Desta forma, o participante que irá iniciar a vídeo-conferência deve pressionar o botão *CALL* que irá disparar um evento solicitando o estabelecimento de uma conexão. Na tela do outro participante, irá aparecer uma janela convidando-o a participar da vídeo-conferência (figura 5.3); caso o participante concorde com o estabelecimento da conexão, imediatamente será aberta uma janela com a imagem e o áudio do participante que iniciou a conexão. Em seguida, o outro usuário aceitando a conexão deve pressionar o botão *Accept* (figura 5.2) para finalizar o processo. A partir deste momento, ambos os participantes estarão recebendo áudio e vídeo estabelecendo-se, desta forma, a vídeo-conferência.

5.3 Passos Para a Construção do ccmVF

A construção do protótipo pode ser dividida em dois passos. O primeiro passo consiste na identificação e especificação dos componentes a serem utilizados. Uma vez especificados

Figura 5.2: Interface do componente de eventos.

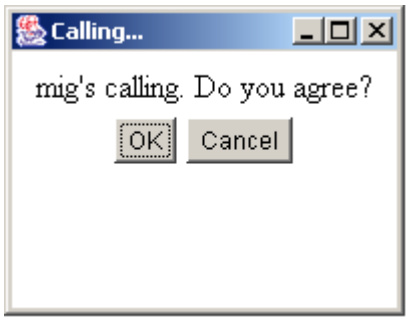

Figura 5.3: Janela de convite.

em XML, os componentes serão submetidos à ferramenta ccmBuilder com a finalidade de gerar os códigos necessários. O segundo passo, consiste na composição dos componentes gerados pela ferramenta ccmBuilder, mais a lógica do programa. A figura 5.4 ilustra os passos necessários para a construção do protótipo que será melhor detalhado nos passos um e dois.

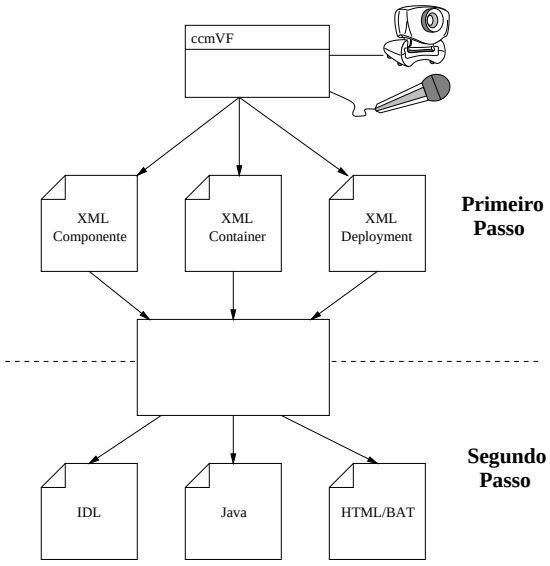

Figura 5.4: Passos para a construção do ccmVF.

5.3.1 Primeiro Passo

No primeiro passo, é feita a escolha de quais componentes irão ser utilizados. Para a construção do ccmVF foi especificado um componente com porta *transmitter* para transmissão de áudio e vídeo e um componente com porta *player* para apresentação de áudio e vídeo. Em adição a estes dois componentes foram especificado mais dois componentes com portas de eventos para controle da aplicação. O componente originador de eventos escolhido possui uma porta *emitter* e o consumidor de eventos possui uma porta *consumer*. A figura 5.5 ilustra os Componentes utilizados no ccmVF e suas ligações.

Depois de definidos os componentes a serem utilizados, torna-se necessário a criação dos arquivos XML. Para cada componente escolhido são gerados três arquivos XML, um descrevendo o componente, outro descrevendo o *container* e um terceiro descrevendo o descritor de distribuição. A ferramenta ccmBuilder irá processar os arquivos XML especificados como mostra a figura 5.6 e irá gerar arquivos Java, IDL e arquivos de distribuição do tipo *bat* para processos ou HTML para *Applet*.

Apresentamos a seguir os arquivos XML gerados para o componente *transmitter*.

Componente *Transmitter*

```
<?xml version="1.0" ?>
<!DOCTYPE componentfeatures
  SYSTEM  "file:///home/mig/windows/CorbaCOS/ccmBldr/dtd/comp.dtd">

<componentfeatures
   name = "AudioGenerator"
   repid = "IDL:AudioGenerator:1.0">

 <ports>
   <!-- Media: Transmits -->
   <transmits
      transmitsname = "AudioTr"
      mediatype = "audio">
   </transmits>
   </ports>  <!-- ports end here -->
</componentfeatures>
```

O componente originador de áudio, neste exemplo, tem o nome *AudioGenerator* e irá disponibilizar a porta *AudioTr*.

Container do Componente *Transmitter*

Um *container* para abrigar o componente anteriormente ilustrado foi criado e pode ser visto no documento XML a seguir.

```
<?xml version="1.0" ?>
```

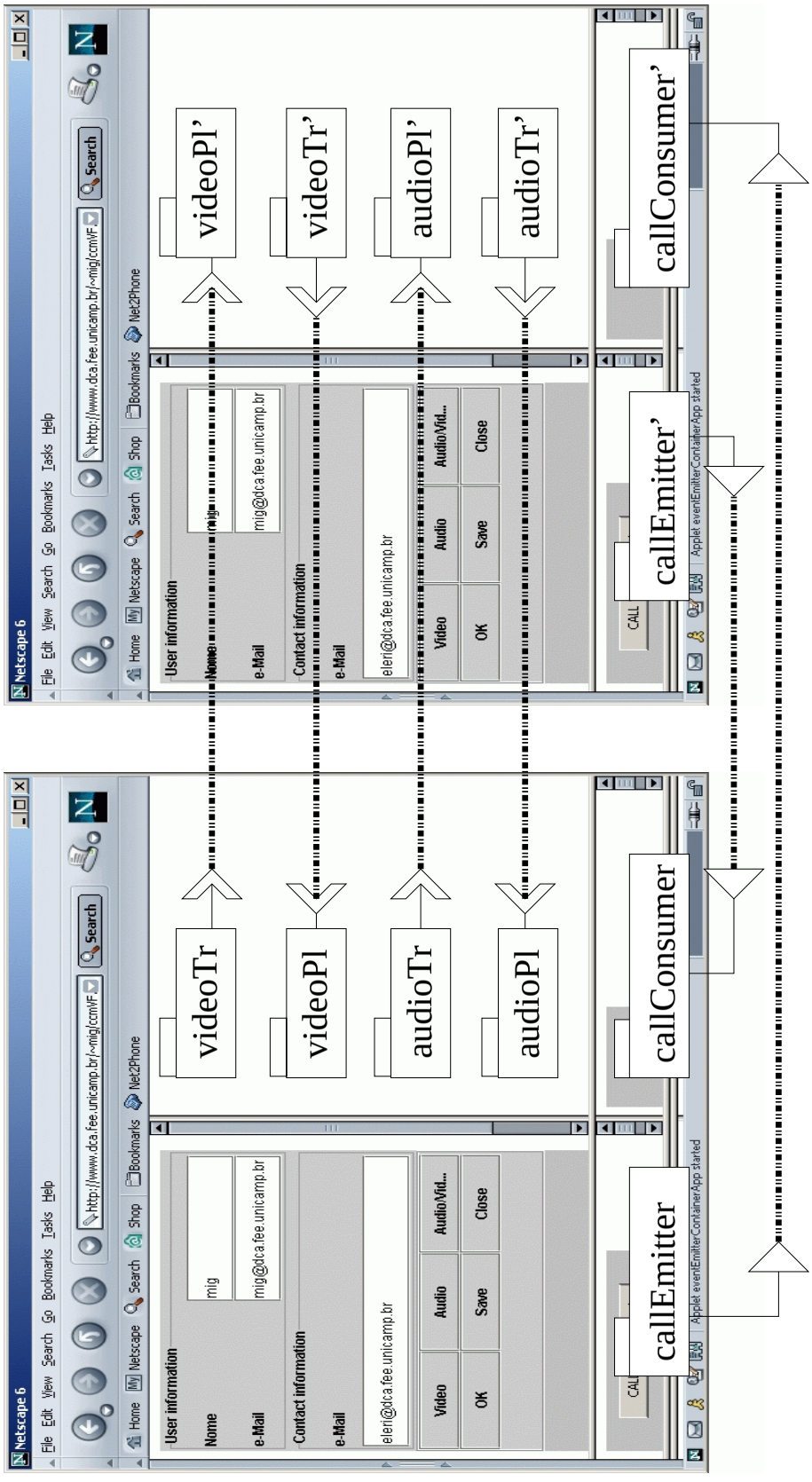

Figura 5.5: Componentes do ccmVF e suas ligações.

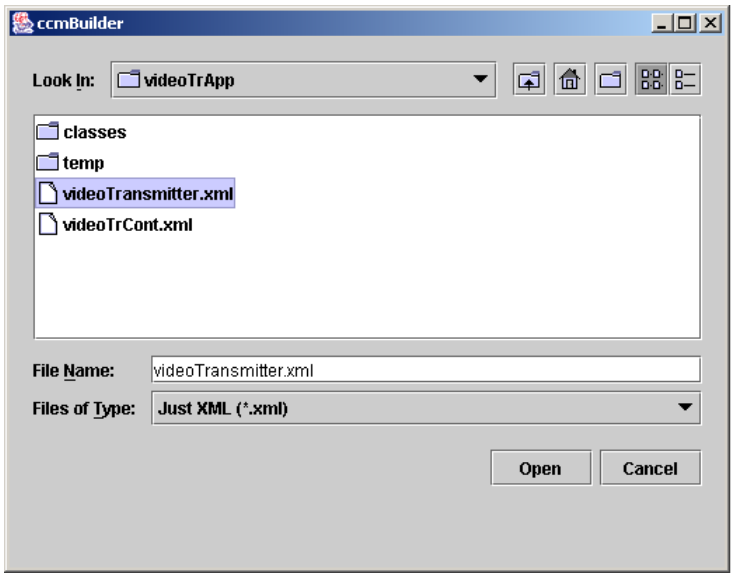

Figura 5.6: Uso do ccmBuilder.

```
<!DOCTYPE container
    SYSTEM  "file:///home/mig/windows/CorbaCOS/ccmBldr/dtd/cont.dtd">
<container
   name = "AudioTransmitter"
   deployedas = "applet">

   <!-- components that will instantiated on this container -->
   <component
      name = "AudioGenerator">
      <!-- dependson name = ""/ -->
   </component>

   <!-- POA -->
   <poapolicies
      thread = "ORB_CTRL_MODEL"
      lifespan = "TRANSIENT"
      iduniqueness = "UNIQUE_ID"
      idassignment = "SYSTEM_ID"
      servantretention = "RETAIN"
      requestprocessing = "USE_ACTIVE_OBJECT_MAP_ONLY"
      implicitactivation = "IMPLICIT_ACTIVATION">
   </poapolicies>
</container>
```

Este *container* terá o nome *AudioTransmitter* que será instalado como uma *Applet* Java.

Descritor de Distribuição do Componente *Transmitter*

Em seguida um descritor de distribuição é especificado em XML como ilustrado a seguir.

```
<?xml version="1.0" ?>
<!DOCTYPE deployment
    SYSTEM "file:///home/mig/windows/CorbaCOS/ccmBldr/dtd/depl.dtd">

<deployment
  class = "AudioTransmitter"
  deployedas = "applet"
  file = "TransmitterAudioTr.html"
  archive = "audioPl.jar,audioTr.jar,jmf.jar,comp.jar,avs.jar,props.jar"
  codebase = "./"
  width = "160"
  height = "160">

  <ORB
    org.omg.CORBA.ORBClass =
       "com.sun.corba.se.internal.POA.POAORB"
    org.omg.CORBA.ORBSingletonClass =
       "com.sun.corba.se.internal.POA.POAORB">
  </ORB>

  <NameService
    NameServiceURL =
      "http://www.dca.fee.unicamp.br/~mig/Tnameserv.html">
  </NameService>

  <HomeFinder
    HomeFinderName = "AudioTransmitter">
  </HomeFinder>
</deployment>
```

O descritor de distribuição quando processado irá gerar um arquivo HTML com o nome de *TransmitterAudioTr.html*. Este arquivo irá conter o nome da classe a ser instanciada (*container*), as bibliotecas necessárias (*jar*), a URL do serviço de nomes, configurações do ORB e o nome do *HomeFinder*.

No Apêndice A, encontram-se todos os arquivos XML necessários para a construção do ccmVF.

5.3.2 Segundo Passo

Depois de gerados os componentes, tornam-se necessário a composição dos mesmos e a inserção de código contendo a lógica da aplicação.

A seguir, iremos detalhar os passos necessários para a composição e inserção de códigos suplementares.

Componente *transmitter*

Comecemos pelo componente transmissor de áudio. Para que ele possa funcionar corretamente, é necessário que sejam feitas algumas alterações nas classes e IDLs (quando necessário) geradas pelo componente. A primeira alteração deve ser feita na fábrica da porta. É gerado pelo ccmBuilder um esqueleto da fábrica da porta que deve ser preenchida com a interface equivalente que é exemplificada a seguir.

```
public AudioGeneratorFactory(AudioGeneratorServant eq, ORB orb, POA poa)
{
   try {
     // --- other ports instantiated here
     AudioTrFactory.create_AudioTr(eq, orb, poa);
   } catch (Exception e) {e.printStackTrace(System.out);}
}
```

O *container* do componente necessita da adição do código exemplificado a seguir, que consiste de instruções para a criação de uma chave que irá identificar o componente, criar o componente e finalmente chamar o método que notifica que o componente foi configurado com sucesso.

```
// ************************************************
// application's code starts here
StringKeyDefaultFactory skf = new StringKeyDefaultFactory();
StringKey skey = skf.init("AudioGenerator" );
System.out.println("StringKey created." );

AudioGenerator comp1 = home_AudioGenerator.create(skey);
System.out.println("Component created.");

// complete configuration
comp1.configuration_complete();
System.out.println("Component configurated.");

// application's code ends here
// ************************************************
```

Conversão de Evento Java para Evento CORBA

Uma característica muito importante do modelo de componentes CM-Tel é a capacidade de converter eventos Java em Eventos CORBA e vice-versa. A conversão de eventos é ilustrada a seguir, onde um evento Java é convertido em um evento CORBA para ser enviado ao consumidor através do ORB.

```
// ActionListener interface
public void actionPerformed (ActionEvent ev) {
  // Code below is a standard way to catch a Java event and
  // transform it to a CORBA event
  // Applications may need to change this code to perform
  // filtering, for instance
  try {
       callRespEvt event = (callRespEvt)(ev.getSource());
       emit_callRespEvt(event);
  } catch(Exception e)
         {System.out.println("Exception in ActionPerformed: "
                             + e.toString());}
}
```

Conexão de Componentes: *Transmitter - Player*

A conexão entre os componentes é realizada no *container* e para melhor compreender como a conexão ocorre, o código a seguir ilustra os passos necessários que são compostos de:

- construção e registro da chave para identificação do componente;
- obtenção da referência do consumidor;
- obtenção da referência da aplicação (*Applet*);
- obtenção da referência do transmissor e;
- finalmente conectar-se ao transmissor.

```
// ************************************************
// application's code starts here

StringKeyDefaultFactory skf = new StringKeyDefaultFactory();
StringKey skey = skf.init("AudioPresenter");
System.out.println("StringKey created.");

AudioPresenter comp1 = home_AudioPresenter.create(skey);
System.out.println("Component created.");

// complete configuration
comp1.configuration_complete();
System.out.println("Component configurated.");

// get the consumer port
Audio.audio_Consumer vc = comp1.provide_AudioPlayer();
```

```
// store applet reference
AppletRef.setAppRef
  (orb.object_to_string(AudioPlayerFactory.getFDev(orb,poa)), this);

// find the Audio Transmitter
AudioGenerator gen_comp =
   AudioGeneratorFinder.find(rootContext, "AudioTransmitter",
                             "AudioGenerator");

AudioTrTransmitter tr = gen_comp.provide_AudioTr();
System.out.println("Transmitter element found.");

// connect to the transmitter
tr.connect_AudioTr(vc);
System.out.println("Connected to transmitter");

// application's code ends here
// ************************************************
```

No Apêndice B, encontram-se o restante dos códigos e suas alterações necessárias ao funcionamento.

5.4 Vantagens da Geração Automática de Código

A utilização do ccmBuilder para gerar a aplicação ccmVF possibilitou um ganho de produção onde 88,8% do código necessário foi gerado automaticamente pela ferramenta. Detalhando melhor, para o componente consumidor de áudio o ganho foi de 93,3%, para o componente produtor de áudio o ganho foi de 97,3%, o consumidor de eventos obteve um ganho de 86,8%, o emissor de eventos conta com 69,8% de código gerado, o consumidor de vídeo 95,5% e o produtor de vídeo 97,8%.

Os arquivos XML necessários para a construção de cada componente não foram levados em conta por serem de tamanho reduzido. Os cálculos foram feitos calculando a quantidade de *bytes* gerada pela ferramenta em relação aos *bytes* necessários para aplicação funcionar.

Por exemplo o consumidor de áudio:

Bytes gerados: 35199

Bytes necessários: 37713

Porcentagem Gerada: 93,33%

Além da vantagem já comprovada, onde, grande parte da aplicação é gerada pela ferramenta, restando ao programador a competência de conectar os componentes e preencher os arquivos com a lógica do programa. Existe ainda a vantagem de que os códigos gerados serem baseados em gabaritos (*templates*) que foram previamente testados aumentando a confiabilidade do código gerado.

5.5 Resumo do Capítulo

Neste capítulo foi apresentado um protótipo que foi construído utilizando-se a ferramenta ccmBuilder bem como os passos necessários para sua construção.

No capítulo seguinte, serão comentados os resultados obtidos e trabalhos futuros.

Capítulo 6

Conclusões e Trabalhos Futuros

Neste trabalho exploramos as tecnologias de desenvolvimento de *software* orientado a componentes e desenvolvemos uma ferramenta para geração automática de código para um modelo de componentes baseado em CORBA, CM-Tel.

Com relação ao escopo e objetivos do trabalho, a seguinte pergunta, norteou o seu desenvolvimento: "*De que forma e com que tecnologia podemos implementar uma ferramenta para suporte ao desenvolvimento de software orientado a componentes?*"

Para tentar responder a essa pergunta, aprofundamo-nos no estudo de novas tecnologias para componentes, no estudo dos documentos e especificações referentes aos modelos de componentes CORBA (CCM) e *Enterprise Java Beans* (EJB), e no modelo CM-Tel, desenvolvido por aluna de doutorado pertencente ao nosso grupo de pesquisa. Também foram estudadas outras tecnologias emergentes na área de computação, como XML, técnicas de *parsing* de documentos XML, e novos elementos da tecnologia CORBA, como o Adaptador de Objetos Portável (POA) e interceptadores portáveis. Com base nos resultados, descritos nos capítulos 3, 4 e 5, podemos considerar que este trabalho atingiu os objetivos propostos.

Alguns pontos que julgamos importante no desenvolvimento da ferramenta ccmBuilder são:

- A geração de código é feita a partir de *templates* que permite um aumento na confiabilidade do *software*, pois os *templates* foram exaustivamente testados.
- Os códigos gerados são aderentes ao padrão de programação CORBA, permitindo, desta forma, a integração entre os componentes gerados e *softwares* legados.
- Suporte completo ao ciclo de desenvolvimento compreendendo a construção do componente, construção do *framework* de componentes, construção do *container* e construção de arquivos de compilação e distribuição.

6.1 Contribuições

Podemos citar como contribuições deste trabalho, os seguintes pontos:

- Construção de uma ferramenta para geração automática de código para o modelo de componentes baseado em CORBA, CM-Tel.
- Desenvolvimento de um *software* para validação do ccmBuilder. Este *software* implementa uma aplicação de vídeo-conferência, que utiliza componentes de produção e consumo de mídias contínuas e componentes de produção e consumo de eventos. Oitenta e dois por cento do código desta aplicação foi gerado pelo ccmBuilder.

6.2 Resultados Obtidos

Os seguintes resultados foram obtidos:

- ccmBuilder vem sendo utilizado em outros trabalhos, como por exemplo o projeto REAL (*REmote Acessible Laboratory*), que é um laboratório virtual acessível pela Internet. O REAL é um projeto desenvolvido pelo CenPRA (Centro de Pesquisas Renato Archer) em conjunto com a Unicamp.
- Publicação como co-autor do trabalho intitulado "Desenvolvimento de Software Orientado a Componentes Para Novos Serviços de Telecomunicações" no 19º Simpósio Brasileiro de Redes de Computadores; Florianópolis-SC, Maio 2001[31].
- Publicação como co-autor do trabalho intitulado "Uma Arquitetura para Disponibilização e Gerência de Serviços na Internet", no 20º Simpósio Brasileiro de Redes de Computadores, Búzios-RJ, Brasil, Maio 2002[32].
- Publicação como co-autor do artigo para revista IEEE Proceedings Issue on Intelligent Mobile Robots Through the Internet intitulado "A Virtual Laboratory Built From Software Components"[6].

6.3 Trabalhos Futuros

Alguns pontos importantes não foram tratados neste trabalho por não estarem dentro de sua proposta inicial. Estes pontos podem ser explorados em trabalhos futuros, dentre os quais podemos citar:

- Utilizar o padrão XSLT[33][34][35][36] (XML Stylesheet Language for Transformations) na geração de código. XSLT permite transformar documentos XML em outros documentos, por exemplo "documentos" Java.
- Geração de código para suporte à qualidade de serviço para os fluxos de mídia contínua de acordo com a tecnologia empregada na rede (DiffServ, RSVP, MPLS, etc). Atualmente, apenas redes DiffServ são suportadas.
- Integrar ccmBuilder com uma ferramenta CASE, por exemplo, Rational Rose. Neste caso, um *parser* converteria os arquivos no formato da ferramenta CASE em arquivos XML compatíveis como ccmBuilder.

A utilização do padrão XSLT bem como a integração do ccmBuilder com uma ferramenta CASE, propostas como trabalhos futuros, hoje se encontram implementados por outro aluno da nossa equipe no ccmBuilder2 como continuidade ao meu trabalho.

Referências Bibliográficas

[1] E. Jendrock M. Pawlan S. Bodoff, D. Green and B. Strearns. *"The J2EE Tutorial"*, 2001. Disponível em `http://java.sun.com/j2ee/tutorial/`.

[2] Object Management Group. "The Common Object Request Broker (CORBA/IIOP)". `http://www.omg.org/technology/`.

[3] Microsoft. "COM: Delivering on the Promises of Component Technology". `http://www.microsoft.com/com/default.asp`.

[4] A. Andoh and S. Nash. "RMI over IIOP - The new RMI-IIOP Standard Features Easy Programming Combined With CORBA connections". `http://www.javaworld.com/javaworld/jw-12-1999/jw-12-iiop.html`.

[5] D. D'Souza and A.C. Wills. *"Objects, Components, and Frameworks with UML: The Catalysis Approach"*. Addison-Wesley, 1999.

[6] E. G. Guimarães, A. T. Maffeis, R. P. Pinto, C. A. Miglinski, E. Cardozo, M. Bergerman, and M. F. Magalhães. "REAL - Virtual Laboratory Built From Software Components". In *IEEE Proceedings Issue on Intelligent Mobile Robots Through the Internet*, 2003.

[7] Rational. "UML Resource Center". `http://www.rational.com/uml/`, 2001.

[8] B. Marchal. *"XML By Example"*. QUE, 1999.

[9] R. Light. *"Iniciando em XML."*. Makron Books, 1999.

[10] Sun Microsystems. *"The Java 2 Enterprise Edition - Developer's Guide - Version 1.2.1"*, Maio. `http://java.sun.com/j2ee/j2sdkee/devguide1_2_1.pdf`.

[11] L. G. DeMichiel et al. *"Enterprise JavaBeans Specification" - Version 2.0*, Agosto 2001. Disponível em `http://java.sun.com`.

[12] H. M. Deitel and P.J. Deitel. *"Java How to Program"*. Prentice Hall, third edition.

[13] Object Management Group. *"CORBA Components". Document orbos/99-02-05*, Marco 1999. Disponível em `http://www.omg.org`.

[14] R. Marvie and P. Merle. *"CORBA Component Model: Discussion and Use with OpenCCM". Document 2001_06_informatica*, Junho 2001. Disponível em `http://openccm.exolab.org`.

[15] The Apache Jakarta Project. "Apache Ant 1.5.1 Manual". `http://jakarta.apache.org/ant/manual/index.html`.

[16] M. Cymerman. "Automate your build process using Java and Ant". `http://www.javaworld.com/javaworld/jw-10-2000/jw-1020-ant.html`.

[17] N. Walsh. "What is XML?". `http://www.xml.com/pub/a/98/10/guide1.html`.

[18] C. Szyperski. *"Components Software: Beyond Object-Oriented Programming"*. Addison-Wesley, Publishing Company, 1997.

[19] C. Szyperski. "Componentes and architectures. Software Development Online, October 2000". `http://www.sdmagazine.com/features/beyound/`.

[20] A. Skonnard and M. Guggin. *"Essencial XML Quick Reference - A programmer's Reference to XML, XPath, XSLT, XML Schema, SOAP, and More"*. Addison-Wesley, 2003.

[21] L. Patterson. Using the IBM XML Parser (XML4J) - Find & Replace Elements in an XML Document. http://www-919.ibm.com/developer/java/xml/xml-parser2.pdf, Janeiro 2000.

[22] W3C. DOM Tutorial. http://www.w3.org/DOM/.

[23] W3School. XML DOM Tutorial. http://www.w3schools.com/.

[24] alphaWorks IBM. "XML Parser for Java". `http://www.alphaworks.ibm.com/tech/xml4j`.

[25] R. Orfali et al. *"The Essencial Distributed Object - Survival Guide"*. John Wiley & Sons, Inc, second edition, 1996.

[26] Software Engineering and System Software Group. "JacORB Home Page". `http://www.jacorb.org`.

[27] ExoLab Group. "OpenORB Home Page". `http://www.openorb.org`.

[28] E. J. Oliveira. "Uma Arquitetura para Acesso a Serviços de Telecomunicações Baseada em TINA". Master's thesis, Faculdade de Engenharia Elétrica e de Computação, Universidade Estadual de Campinas, Brasil, 1999.

[29] Object Management Group. *"Property Service Specification" - Version 1.0 - Specification formal/2000-06-22*, Abril 2002. Disponível em `http://www.omg.org`.

[30] Object Management Group. *"CORBA Services"*, 2001. Disponível em `http://www.omg.org`.

[31] E. G. Guimarães, A. T. Maffeis, J. L. Pereira, B. G. Russo, M. Bergerman, E. Cardoso, M. F. Magalhães, C. A. Miglinski, and R. P. Pinto. "Desenvolvimento de Software Orientado a Componentes Para Novos Serviços de Telecomunicações". In *19º Simpósio Brasileiro de Redes de Computadores (SBRC 2001)*, Florianópolis-SC, Brasil, Maio 2001.

[32] R. P. Pinto, L. F. Faina, A. T. Maffeis, E. G. Guimarães, C. A. Miglinski, and E. Cardozo. "Uma Arquitetura para Disponibilização e Gerência de Serviços na Internet". In *20º Simpósio Brasileiro de Redes de Computadores (SBRC 2002)*, Buzios-RJ, Brasil, Maio 2002.

[33] E. Armstrong et al. "The Java Web Services Tutorial". `http://java.sun.com/webservices/docs/1.0/tutorial/index.html`.

[34] E. Armstrong. "Working with XML - The Java API for XML Processing (JAXP) Tutorial". `http://java.sun.com/xml/jaxp/dist/1.1/docs/tutorial/index.html`.

[35] K. Cagle. *"XML - Developer's Handbook."*. Sybex, 2000.

[36] A. Homer. *"XML IE5 - Programmer's Reference"*. Wrox Press ltd, 1999.

Apêndice A

Componentes, Container e Deployments do ccmVF

A.1 Áudio *Player*

A.1.1 XML do Componente Áudio *Player*

```
<?xml version="1.0" ?>
<!DOCTYPE componentfeatures
   SYSTEM "file:///home/mig/windows/CorbaCOS/ccmBldr/dtd/comp.dtd">
<componentfeatures
   name = "AudioPresenter"
   repid = "IDL:AudioPresenter:1.0">
 <ports>
   <!-- Media: Player -->
   <plays
      playsname = "AudioPlayer"
      mediatype = "audio">
   </plays>
 </ports>  <!-- ports end here -->
</componentfeatures>
```

A.1.2 XML do *Container* para o Componente Áudio *Player*

```
<?xml version="1.0" ?>
<!DOCTYPE container
    SYSTEM  "file:///home/mig/windows/CorbaCOS/ccmBldr/dtd/cont.dtd">
<container
   name = "AudioPlayer"
   deployedas = "applet">

   <!-- components that will instantiated on this container -->
```

```
   <component
      name = "AudioPresenter">
      <dependson name = "AudioGenerator"/>
   </component>

   <!-- POA -->
   <poapolicies
      thread = "ORB_CTRL_MODEL"
      lifespan = "TRANSIENT"
      iduniqueness = "UNIQUE_ID"
      idassignment = "SYSTEM_ID"
      servantretention = "RETAIN"
      requestprocessing = "USE_ACTIVE_OBJECT_MAP_ONLY"
      implicitactivation = "IMPLICIT_ACTIVATION">
   </poapolicies>
</container>
```

A.1.3 Descritor de Distribuição do Componente Áudio *Player*

```
<?xml version="1.0" ?>
<!DOCTYPE deployment
    SYSTEM "file:///home/mig/windows/CorbaCOS/ccmBldr/dtd/depl.dtd">

<deployment
  class = "AudioPlayer"
  deployedas = "applet"
  file = "PlayerAudioTr.html"
  archive = "audioPl.jar,audioTr.jar,jmf.jar,comp.jar,avs.jar,props.jar"
  codebase = "./"
  width = "160"
  height = "160">

<ORB
  org.omg.CORBA.ORBClass = "com.sun.corba.se.internal.POA.POAORB"
  org.omg.CORBA.ORBSingletonClass = "com.sun.corba.se.internal.POA.POAORB">
</ORB>

<NameService
  NameServiceURL = "http://www.dca.fee.unicamp.br/~mig/Tnameserv.html">
</NameService>

<HomeFinder
```

```
  HomeFinderName = "AudioPlayer">
</HomeFinder>

<AVStreams>
  <AudioQoS
    audio_sampleRate = "8000"
    audio_sampleSize = "8"
    audio_numChannels = "1"
    audio_quantisation = "1">
  </AudioQoS>
</AVStreams>

</deployment>
```

A.2 Áudio *Transmitter*

A.2.1 XML do Componente Áudio *Transmitter*

```
<?xml version="1.0" ?>
<!DOCTYPE componentfeatures
  SYSTEM  "file:///home/mig/windows/CorbaCOS/ccmBldr/dtd/comp.dtd">
<componentfeatures
   name = "AudioGenerator"
   repid = "IDL:AudioGenerator:1.0">
 <ports>
   <!-- Media: Transmits -->
   <transmits
      transmitsname = "AudioTr"
      mediatype = "audio">
   </transmits>
   </ports>  <!-- ports end here -->
</componentfeatures>
```

A.2.2 XML do *Container* para o Componente Áudio *Transmitter*

```
<?xml version="1.0" ?>
<!DOCTYPE container
    SYSTEM  "file:///home/mig/windows/CorbaCOS/ccmBldr/dtd/cont.dtd">
<container
   name = "AudioTransmitter"
   deployedas = "applet">
   <!-- components that will instantiated on this container -->
```

```
   <component
      name = "AudioGenerator">
      <!-- dependson name = ""/ -->
   </component>
   <!-- POA -->
   <poapolicies
      thread = "ORB_CTRL_MODEL"
      lifespan = "TRANSIENT"
      iduniqueness = "UNIQUE_ID"
      idassignment = "SYSTEM_ID"
      servantretention = "RETAIN"
      requestprocessing = "USE_ACTIVE_OBJECT_MAP_ONLY"
      implicitactivation = "IMPLICIT_ACTIVATION">
   </poapolicies>
</container>
```

A.2.3 Descritor de Distribuição do Componente Áudio *Transmitter*

```
<?xml version="1.0" ?>
<!DOCTYPE deployment
    SYSTEM "file:///home/mig/windows/CorbaCOS/ccmBldr/dtd/depl.dtd">
<deployment
  class = "AudioTransmitter"
  deployedas = "applet"
  file = "TransmitterAudioTr.html"
  archive = "audioPl.jar,audioTr.jar,jmf.jar,comp.jar,avs.jar,props.jar"
  codebase = "./"
  width = "160"
  height = "160">
  <ORB
    org.omg.CORBA.ORBClass = "com.sun.corba.se.internal.POA.POAORB"
    org.omg.CORBA.ORBSingletonClass = "com.sun.corba.se.internal.POA.POAORB">
  </ORB>
  <NameService
    NameServiceURL = "http://www.dca.fee.unicamp.br/~mig/Tnameserv.html">
  </NameService>
  <HomeFinder
    HomeFinderName = "AudioTransmitter">
  </HomeFinder>
  <AVStreams>
    <AudioQoS
      audio_sampleRate = "8000"
      audio_sampleSize = "8"
```

```
      audio_numChannels = "1"
      audio_quantisation = "1">
    </AudioQoS>
  </AVStreams>
</deployment>
```

A.3 Consumidor de Eventos

A.3.1 XML do Componente Consumidor de Eventos

```
<?xml version="1.0" ?>
<!DOCTYPE componentfeatures
  SYSTEM "file:///home/mig/windows/CorbaCOS/ccmBldr/dtd/comp.dtd">
<componentfeatures
   name = "evtPresenter"
   repid = "IDL:evtPresenter:1.0">
 <ports>
   <!-- Events: Consumer -->
   <consumes
      consumesname = "ccmConsumer"
      eventtype = "callRespEvt">
      <eventpolicy policy = "normal"/>
   </consumes>
   </ports>  <!-- ports end here -->
</componentfeatures>
```

A.3.2 XML do *Container* para o Componente Consumidor de Eventos

```
<?xml version="1.0" ?>
<!DOCTYPE container
   SYSTEM "file:///home/mig/windows/CorbaCOS/ccmBldr/dtd/cont.dtd">
<container
   name = "eventConsumer"
   deployedas = "applet">
   <!-- components that will instantiated on this container -->
   <component
      name = "evtPresenter">
      <dependson name = "evtGenerator"/>
   </component>
   <!-- POA -->
   <poapolicies
      thread = "ORB_CTRL_MODEL"
```

```
      lifespan = "TRANSIENT"
      iduniqueness = "UNIQUE_ID"
      idassignment = "SYSTEM_ID"
      servantretention = "RETAIN"
      requestprocessing = "USE_ACTIVE_OBJECT_MAP_ONLY"
      implicitactivation = "IMPLICIT_ACTIVATION">
   </poapolicies>
</container>
```

A.3.3 Descritor de Distribuição do Componente Consumidor de Eventos

```
<?xml version="1.0" ?>
<!DOCTYPE deployment
    SYSTEM "file:///home/mig/windows/CorbaCOS/ccmBldr/dtd/depl.dtd">
<deployment
  class = "eventConsumer"
  deployedas = "applet"
  file = "Cons.html"
  archive = "emt.jar,consumer.jar,evts.jar,comp.jar"
  codebase = "./"
  width = "160"
  height = "160">
  <ORB
    org.omg.CORBA.ORBClass = "com.sun.corba.se.internal.POA.POAORB"
    org.omg.CORBA.ORBSingletonClass = "com.sun.corba.se.internal.POA.POAORB">
  </ORB>
  <NameService
    NameServiceURL = "http://www.dca.fee.unicamp.br/~mig/Tnameserv.html">
  </NameService>
  <HomeFinder
    HomeFinderName = "Consumer">
  </HomeFinder>
</deployment>
```

A.4 Originador de Eventos

A.4.1 XML do Componente Originador de Eventos

```
<?xml version="1.0" ?>
<!DOCTYPE componentfeatures
  SYSTEM "file:///home/mig/windows/CorbaCOS/ccmBldr/dtd/comp.dtd">
<componentfeatures
```

```
   name = "evtGenerator"
   repid = "IDL:evtGenerator:1.0">
 <ports>
   <!-- Events: Emitter -->
   <emits
      emitsname = "ccmEvt"
      eventtype = "callRespEvt">
      <eventpolicy policy = "normal"/>
   </emits>
   </ports>  <!-- ports end here -->
</componentfeatures>
```

A.4.2 XML do *Container* para o Componente Originador de Eventos

```
<?xml version="1.0" ?>
<!DOCTYPE container
    SYSTEM  "file:///home/mig/windows/CorbaCOS/ccmBldr/dtd/cont.dtd">
<container
   name = "eventEmitter"
   deployedas = "applet">
   <!-- components that will instantiated on this container -->
   <component
      name = "evtGenerator">
      <dependson name = "evtGenerator"/>
   </component>
   <!-- POA -->
   <poapolicies
      thread = "ORB_CTRL_MODEL"
      lifespan = "TRANSIENT"
      iduniqueness = "UNIQUE_ID"
      idassignment = "SYSTEM_ID"
      servantretention = "RETAIN"
      requestprocessing = "USE_ACTIVE_OBJECT_MAP_ONLY"
      implicitactivation = "IMPLICIT_ACTIVATION">
   </poapolicies>
</container>
```

A.4.3 Descritor de Distribuição do Componente Originador de Eventos

```
<?xml version="1.0" ?>
<!DOCTYPE deployment
```

```
    SYSTEM "file:///home/mig/windows/CorbaCOS/ccmBldr/dtd/depl.dtd">
<deployment
  class = "eventEmitter"
  deployedas = "applet"
  file = "Emt.html"
  archive = "emt.jar,consumer.jar,evts.jar,comp.jar"
  codebase = "./"
  width = "160"
  height = "160">
  <ORB
    org.omg.CORBA.ORBClass = "com.sun.corba.se.internal.POA.POAORB"
    org.omg.CORBA.ORBSingletonClass = "com.sun.corba.se.internal.POA.POAORB">
  </ORB>
  <NameService
    NameServiceURL = "http://www.dca.fee.unicamp.br/~mig/Tnameserv.html">
  </NameService>
  <HomeFinder
    HomeFinderName = "Emitter">
  </HomeFinder>
</deployment>
```

A.5 Vídeo *Player*

A.5.1 XML do Componente Vídeo *Player*

```
<?xml version="1.0" ?>
<!DOCTYPE componentfeatures
    SYSTEM "file:///home/mig/windows/CorbaCOS/ccmBldr/dtd/comp.dtd">
<componentfeatures
   name = "VideoPresenter"
   repid = "IDL:VideoPresenter:1.0">
 <ports>
   <!-- Media: Player -->
   <plays
      playsname = "VideoPlayer"
      mediatype = "video">
   </plays>
 </ports>  <!-- ports end here -->
</componentfeatures>
```

A.5.2 XML do *Container* para o Componente Video *Player*

```
<?xml version="1.0" ?>
```

```
<!DOCTYPE container
    SYSTEM  "file:///home/mig/windows/CorbaCOS/ccmBldr/dtd/cont.dtd">
<container
   name = "VideoPlayer"
   deployedas = "applet">
   <!-- components that will instantiated on this container -->
   <component
      name = "VideoPresenter">
      <dependson name = " VideoTRGenerator"/>
   </component>
   <!-- POA -->
   <poapolicies
      thread = "ORB_CTRL_MODEL"
      lifespan = "TRANSIENT"
      iduniqueness = "UNIQUE_ID"
      idassignment = "SYSTEM_ID"
      servantretention = "RETAIN"
      requestprocessing = "USE_ACTIVE_OBJECT_MAP_ONLY"
      implicitactivation = "IMPLICIT_ACTIVATION">
   </poapolicies>
</container>
```

A.5.3 Descritor de Distribuição do Componente Video *Player*

```
<?xml version="1.0" ?>
<!DOCTYPE deployment
    SYSTEM "file:///home/mig/windows/CorbaCOS/ccmBldr/dtd/depl.dtd">
<deployment
  class = "VideoPlayer"
  deployedas = "applet"
  file = "PlayerVideoTr.html"
  archive = "videoPl.jar,videoTr.jar,jmf.jar,comp.jar,avs.jar,props.jar"
  codebase = "./"
  width = "160"
  height = "160">
  <ORB
    org.omg.CORBA.ORBClass = "com.sun.corba.se.internal.POA.POAORB"
    org.omg.CORBA.ORBSingletonClass = "com.sun.corba.se.internal.POA.POAORB">
  </ORB>
  <NameService
    NameServiceURL = "http://www.dca.fee.unicamp.br/~mig/Tnameserv.html">
  </NameService>
  <HomeFinder
    HomeFinderName = "VideoPlayer">
```

```
  </HomeFinder>
  <AVStreams>
   <VideoQoS
      video_framerate = "25"
      video_colorDepth = "24"
      video_colorModel = "1"
      video_resolution = "320X240">
   </VideoQoS>
  </AVStreams>
</deployment>
```

A.6 Vídeo *Transmitter*

A.6.1 XML do Componente Vídeo *Transmitter*

```
<?xml version="1.0" ?>
<!DOCTYPE componentfeatures
    SYSTEM "file:///home/mig/windows/CorbaCOS/ccmBldr/dtd/comp.dtd">
<componentfeatures
   name = "VideoTRGenerator"
   repid = "IDL:VideoTRGenerator:1.0">
 <ports>
   <!-- Media: Transmits -->
   <transmits
      transmitsname = "VideoTr"
      mediatype = "video">
   </transmits>
   </ports>  <!-- ports end here -->
</componentfeatures>
```

A.6.2 XML do *Container* para o Componente Vídeo *Transmitter*

```
<?xml version="1.0" ?>
<!DOCTYPE container
    SYSTEM  "file:///home/mig/windows/CorbaCOS/ccmBldr/dtd/cont.dtd">
<container
   name = "VideoTransmitter"
   deployedas  = "applet">
   <!-- components that will instantiated on this container -->
   <component
      name = "VideoTRGenerator">
      <!-- dependson name = ""/ -->
```

```
   </component>
   <!-- POA -->
   <poapolicies
      thread = "ORB_CTRL_MODEL"
      lifespan = "TRANSIENT"
      iduniqueness = "UNIQUE_ID"
      idassignment = "SYSTEM_ID"
      servantretention = "RETAIN"
      requestprocessing = "USE_ACTIVE_OBJECT_MAP_ONLY"
      implicitactivation = "IMPLICIT_ACTIVATION">
   </poapolicies>
</container>
```

A.6.3 Descritor de Distribuição do Componente Vídeo *Transmitter*

```
<?xml version="1.0" ?>
<!DOCTYPE deployment
    SYSTEM "file:///home/mig/windows/CorbaCOS/ccmBldr/dtd/depl.dtd">
<deployment
  class = "VideoTransmitter"
  deployedas = "applet"
  file = "TransmitterVideoTr.html"
  archive = "videoPl.jar,videoTr.jar,jmf.jar,comp.jar,avs.jar,props.jar"
  codebase = "./"
  width = "160"
  height = "160">
  <ORB
    org.omg.CORBA.ORBClass = "com.sun.corba.se.internal.POA.POAORB"
    org.omg.CORBA.ORBSingletonClass = "com.sun.corba.se.internal.POA.POAORB">
  </ORB>
  <NameService
    NameServiceURL = "http://www.dca.fee.unicamp.br/~mig/Tnameserv.html">
  </NameService>
  <HomeFinder
    HomeFinderName = "VideoTransmitter">
  </HomeFinder>
  <AVStreams>
   <VideoQoS
      video_framerate = "25"
      video_colorDepth = "24"
      video_colorModel = "1"
      video_resolution = "320X240">
   </VideoQoS>
```

```
  </AVStreams>
</deployment>
```

Apêndice B

Alterações Necessárias aos Arquivos Java e IDLs do ccmVF

B.1 Consumidor de Áudio

B.1.1 Fábrica do Componente Conmsumidor de Áudio

```
public class AudioPresenterFactory {

    public AudioPresenterFactory(AudioPresenterServant eq, ORB orb, POA poa)
    {
    try {
// --- other ports instantiated here
AudioPlayerFactory.create_AudioPlayer(eq, orb, poa);

        } catch (Exception e) {e.printStackTrace(System.out);}
    }
}
```

B.1.2 *Container* do Componente Consumidor de Áudio

```
// ************************************************
// application's code starts here

StringKeyDefaultFactory skf = new StringKeyDefaultFactory();
StringKey skey = skf.init("AudioPresenter"+usrMail);
System.out.println("StringKey created. =====> AudioPresenter"+usrMail);

AudioPresenter comp1 = home_AudioPresenter.create(skey);
System.out.println("Component created.");

// complete configuration
```

```
comp1.configuration_complete();
System.out.println("Component configurated.");

// get the consumer port
Audio.audio_Consumer vc = comp1.provide_AudioPlayer();


// store applet reference
AppletRef.setAppRef
  (orb.object_to_string(AudioPlayerFactory.getFDev(orb,poa)), this);

// find the Audio Transmitter
AudioGenerator gen_comp =
  AudioGeneratorFinder.find(rootContext, "AudioTransmitter",
  "AudioGenerator"+
  contMail);

AudioTrTransmitter tr = gen_comp.provide_AudioTr();
System.out.println("Transmitter element found.");

// connect to the broadcaster
tr.connect_AudioTr(vc);
System.out.println("Connected to transmitter");

// application's code ends here
// ************************************************
```

B.2 Produtor de Áudio

B.2.1 Fábrica do Componente Produtor de Áudio

```
public class AudioGeneratorFactory {

    public AudioGeneratorFactory(AudioGeneratorServant eq, ORB orb, POA poa)
    {
      try {
// --- other ports instantiated here
AudioTrFactory.create_AudioTr(eq, orb, poa);
      } catch (Exception e) {e.printStackTrace(System.out);}
    }
}
```

B.2.2 *Container* do Componente Consumidor de Áudio

```
// ************************************************
// application's code starts here
StringKeyDefaultFactory skf = new StringKeyDefaultFactory();
StringKey skey = skf.init("AudioGenerator" +usrMail);
System.out.println("StringKey created. =====> AudioGenerator"+usrMail);

AudioGenerator comp1 = home_AudioGenerator.create(skey);
System.out.println("Component created.");

// complete configuration
comp1.configuration_complete();
System.out.println("Component configurated.");

// application's code ends here
// ************************************************
```

B.3 Consumidor de Eventos

B.3.1 IDL do Evento

```
valuetype callRespEvt //: Components::EventBase
 {
    // complete event definition here
   public string ccm_Evt;

   //initializer
   factory init (in string _ccm_Evt);
 };
```

B.3.2 Implementação do Objeto por Valor do Eventos

```
public class callRespEvtImpl extends callRespEvt {

  public callRespEvtImpl() {
  // complete event initialization here (default values)
    ccm_Evt = new String("");
  }

  // add here the factory with the proper initialization parameters
    public callRespEvtImpl(String _ccm_Evt) {
ccm_Evt = new String(_ccm_Evt);
    }
```

```
}
```

B.3.3 Fábrico do Componente Consumidor de Eventos

```
public class evtPresenterFactory {

    public evtPresenterFactory(evtPresenterServant eq, ORB orb, POA poa)
    {
    try {
// --- other ports instantiated here
ccmConsumerConsumer ccc =
    ccmConsumerFactory.create_ccmConsumer(eq, orb,poa);

        } catch (Exception e) {e.printStackTrace(System.out);}
    }
}
```

B.3.4 *Container* do Componente Consumidor de Eventos

```
// ************************************************
// application's code starts here
// create an instance of URLPresenter
StringKeyDefaultFactory skf = new StringKeyDefaultFactory();
StringKey skey = skf.init("evtPresenter"+usrMail);
System.out.println("StringKey created."+"evtPresenter"+usrMail);

evtPresenter comp1 = home_evtPresenter.create(skey);
System.out.println("Component created.");

// complete configuration
comp1.configuration_complete();
System.out.println("Component configurated.");

// get the URLConsumer element
ccmConsumerConsumer cons = comp1.provide_ccmConsumer();
System.out.println("Consumer accessed.");

ccmConsumerConsumerServant cons_serv =
  (ccmConsumerConsumerServant)poa.reference_to_servant(cons);

// add this applet as a listener
cons_serv.addActionListener(new callRespEventListenerApp(this));

// enable events
```

```
cons_serv.enableEvents();

try {
     gen_comp =
     evtGeneratorFinder.find
 (rootContext,"Emitter","evtGenerator"+contMail);
     System.out.println("Find =====> Emitter"+
       "  evtGenerator"+contMail)
} catch(Exception e) {
   System.out.println("No emitter running");
   gen_comp = null;
   System.exit(0);
}
try {
  // find component
  System.out.println("Generator component found (eq. int.).");

  // find emitter port
  emt = gen_comp.provide_ccmEvt();
  System.out.println("Emitter element found.");
} catch (Exception e ) { gen_comp = null;}

// connect to the emitter
emt.connect_ccmEvt(cons);
System.out.println("Connected to emitter  ");

// application's code ends here
// ************************************************
```

B.3.5 Aplicação que Utiliza o Evento

```
java.lang.Object obj = ev.getSource();
callRespEvt evtStr = (callRespEvt)obj;
String msg = new String();
msg = evtStr.ccm_Evt;
if (!msg.equals("Accept")) {
  toggleForm();
   l.setText(evtStr.ccm_Evt + ". Do you agree?");
   System.out.println("Arriving " + evtStr.ccm_Evt);
}
else
{
try {
  URL url = new URL
```

```
            ("http://www.dca.fee.unicamp.br/~mig/ccmVFPl.html");
  ctx.showDocument(url,"ccmPl");
} catch (Exception e ) {System.out.println(e);}
```

B.4 Produtor de Evento

B.4.1 IDL do Evento

```
valuetype callRespEvt //: Components::EventBase
 {
    // complete event definition here
   public string ccm_Evt;

   //initializer
   factory init (in string _ccm_Evt);
 };
\end{verbaim}

\subsection{Implementação do Objeto por Valor do Eventos}
\begin{verbatim}
public class callRespEvtImpl extends callRespEvt {

  public callRespEvtImpl() {
    // complete event initialization here (default values)
    ccm_Evt = new String("");
  }

  // add here the factory with the proper initialization parameters
  public callRespEvtImpl(String _ccm_Evt) {
    ccm_Evt = new String(_ccm_Evt);
  }
}
```

B.4.2 Fábrica do Componente Produtor de Eventos

```
public class evtGeneratorFactory {
  public evtGeneratorFactory(evtGeneratorServant eq, ORB orb, POA poa)
  {
    try {
      // --- other ports instantiated here
ccmEvtEmitter cee =
    ccmEvtFactory.create_ccmEvt(eq,orb,poa);
```

```
    } catch (Exception e) {e.printStackTrace(System.out);}
  }
}
```

B.4.3 *Container* do Componente Produtor de Eventos

```
// ************************************************
// application's code starts here

// create an instance of URLGenerator
StringKeyDefaultFactory skf = new StringKeyDefaultFactory();
StringKey skey = skf.init("evtGenerator"+usrMail);
System.out.println("StringKey created."+"evtGenerator"+usrMail);

evtGenerator comp1 = home_evtGenerator.create(skey);
System.out.println("Component created.");

// complete configuration
comp1.configuration_complete();
System.out.println("Component configurated.");

// get the ccmEvt element
ccmEvtEmitter evt = comp1.provide_ccmEvt();
System.out.println("Emitter accessed.");

// get a reference of the ccmEvtEmitterServant
ccmEvtEmitterServant emtServ =
   (ccmEvtEmitterServant)poa.reference_to_servant(evt);
System.out.println("Got reference of emitter.");

PhoneAsk pa = new PhoneAsk(usrName,this);
add(pa);

// connect to the event source
pa.addActionListener(emtServ);
setVisible(true);
// application's code ends here
// ************************************************
```

B.4.4 Classe que Utiliza o Evento

```
public void actionPerformed(ActionEvent evt) {
  if (evt.getActionCommand().equals("CALL")) {
if (MediaType.equalsIgnoreCase("VIDEO"))
```

```
Msg.setText("(VIDEO) Calling!!");
    else
Msg.setText("(AUDIO) Calling!!");

     callRespEvt callResp = fact.init(usrName+"\'s Calling");
     ActionListener al = null;
     ActionEvent ev = new ActionEvent(callResp,0,usrName+"\'s Calling");
     Vector actionListeners = getListeners();
     for (int i = 0;i < actionListeners.size();i++) {
        al = (ActionListener)actionListeners.elementAt(i);
al.actionPerformed(ev);
     }

  }
  if (evt.getActionCommand().equals("Accept")) {
      callRespEvt callResp = fact.init("Accept");
      ActionListener al = null;
      ActionEvent ev = new ActionEvent(callResp,0,"Accept");
      Vector actionListeners = getListeners();
      for (int i = 0;i < actionListeners.size();i++) {
al = (ActionListener)actionListeners.elementAt(i);
al.actionPerformed(ev);
      }
   }
```

B.5 Consumidor de Vídeo

B.5.1 Fábrica do Componente Consumidor de Vídeo

```
public class VideoPresenterFactory {
    public VideoPresenterFactory(VideoPresenterServant eq, ORB orb, POA poa)
    {
    try {
// --- other ports instantiated here
VideoPlayerFactory.create_VideoPlayer(eq, orb, poa);

        } catch (Exception e) {e.printStackTrace(System.out);}
    }
}
```

B.5.2 *Container* do Componente Consumidor de Vídeo

```
// ************************************************
```

```
// application's code starts here

StringKeyDefaultFactory skf = new StringKeyDefaultFactory();
StringKey skey = skf.init("VideoPresenter"+usrMail);
System.out.println("StringKey created.");

VideoPresenter comp1 = home_VideoPresenter.create(skey);
System.out.println("Component created.");

// complete configuration
comp1.configuration_complete();
System.out.println("Component configurated.");

// get the consumer port
Video.video_Consumer vc = comp1.provide_VideoPlayer();

// store applet reference
AppletRef.setAppRef
  (orb.object_to_string(VideoPlayerFactory.getFDev(orb,poa)), this);

// find the Video Transmitter
 VideoTRGenerator gen_comp = VideoTRGeneratorFinder.find(
     rootContext, "VideoTransmitter", "VideoGenerator"+
     contMail);

VideoTrTransmitter tr = gen_comp.provide_VideoTr();
System.out.println("Transmitter element found.");

// connect to the broadcaster
tr.connect_VideoTr(vc);
System.out.println("Connected to transmitter");

// application's code ends here
// ************************************************
```

B.6 Produtor de Vídeo

B.6.1 Fábrica do Componente Produtor de Vídeo

```
public class VideoTRGeneratorFactory {
  public VideoTRGeneratorFactory
      (VideoTRGeneratorServant eq, ORB orb, POA poa)
  {
```

```
    try {
      // --- other ports instantiated here
      VideoTrFactory.create_VideoTr(eq, orb, poa);

     } catch (Exception e) {e.printStackTrace(System.out);}
  }
}
```

B.6.2 *Container* do Componente Produtor de Vídeo

```
// ************************************************
// application's code starts here

StringKeyDefaultFactory skf = new StringKeyDefaultFactory();
StringKey skey = skf.init("VideoGenerator"+usrMail);
System.out.println("StringKey created.");

VideoTRGenerator comp1 = home_VideoTRGenerator.create(skey);
System.out.println("Component created.");

// complete configuration
comp1.configuration_complete();
System.out.println("Component configurated.");

// application's code ends here
// ************************************************
```