MAC 5701 - Tópicos em Ciência da Computação

Introdução ao CORBA Component Model (CCM)

Alexandre Ricardo Nardi
Orientador: Prof. Francisco C. R. Reverbel

Junho de 2001

Resumo. Desde sua criação, a arquitetura CORBA tem sido cada vez mais utilizada para comunicação entre objetos, e evoluído com a especificação de novas funcionalidades e padrões. O CORBA Component Model faz parte da especificação do CORBA 3.0, e traz significativas melhoras na organização dos objetos em componentes, facilitando o trabalho de modelagem, desenvolvimento, empacotamento, implantação e execução.

1. Introdução

Desde sua concepção, em 1988, pela Objects Management Group (OMG), CORBA tem se tornado padrão e praticamente sinônimo de flexibilidade e versatilidade quando se trata de comunicação entre objetos, independentemente de linguagem de programação, sistema operacional e mesmo protocolo de rede. Empresas como IONA, Inprise e BEA Systems desenvolveram e lançaram, a partir de então, suas implementações comerciais de ORB´s, cada qual com suas particularidades, mas sempre procurando respeitar o padrão definido, e contribuindo para que este evoluísse, com conceitos como o Portable Object Adapter (POA), mecanismo do ORB responsável por encaminhar as requisições clientes para as implementações dos objetos, que substituiu o então Basic Object Adapter (BOA), trazendo mais robustez ao CORBA, uma vez que fornece API´s para registrar as implementações dos objetos com o ORB, desativar objetos ou ainda ativá-los sob demanda.

Outro fator que tem contribuído com a difusão do CORBA é a popularização da linguagem Java, possibilitando comunicação entre objetos em ambiente distribuído, até mesmo via Internet, dado que o Java Plug-In inclui implementação de ORB nativa, facilitando muito a distribuição de aplicações.

Este, entretanto, é um padrão em constante evolução e amadurecimento, e a próxima versão do mesmo, CORBA 3.0, agrega novas funcionalidades que se propõem a resolver situações hoje desconfortáveis para o desenvolvedor e pelos responsáveis pela manutenção de aplicações, como veremos a seguir.

Problemas

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

A especificação do CORBA 2.3 [OMG99] apresenta algumas falhas [WSO00, CCM99], como:

- Falta de um padrão para o empacotamento e a implantação (*packaging and deployment*) dos objetos, o que agrega risco a projetos mais complexos, que utilizem centenas ou milhares de objetos;
- Falta de suporte para o uso de subconjuntos de funcionalidades “comuns”. Por exemplo, um grande número de aplicações utiliza apenas um subconjunto das possíveis configurações do POA e, apesar disso, o desenvolvedor deve conhecer muitas políticas do POA para conseguir o efeito desejado, o que possui o efeito colateral de aumentar a complexidade de modo a comprometer a produtividade daquele. Isso tem feito com que haja busca por padrões que se repitam com freqüência no desenvolvimento de sistemas, agilizando o desenvolvimento e a utilização de ferramentas de geração de código;
- Dificuldade para estender funcionalidades de objetos CORBA, possível apenas através de herança. Para suportar novas interfaces, o desenvolvedor deve definir nova interface em IDL que herde de todas as interfaces em questão, implemente tal interface e implante a nova implementação nos servidores. Entretanto, a herança múltipla, possível em CORBA IDL, é frágil, pois há linguagens que não dão suporte a redefinição (*overloading*), como C, por exemplo;
- Falta de definição de serviços obrigatórios: a aplicação deve possuir código que trate a indisponibilidade de serviços, como no caso do

gerenciamento de ciclo-de-vida de objetos: apesar da existência do "*Lifecycle Service*", seu uso não é obrigatório, o que dificulta o trabalho no cliente;

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

Visão Geral

Com o objetivo de resolver estes e outros problemas, o CORBA 3.0 traz a especificação do CORBA Component Model (CCM), que se utiliza do conceito de componentes para agregar objetos afins, sendo gerenciados por estruturas denominadas containers, com comportamento padronizado, sem a perda de flexibilidade e versatilidade do CORBA. Os componentes trazem consigo funcionalidades como publicação e subscrição de eventos, que facilitam a administração, bem como novo padrão para empacotamento e distribuição, aliviando sobremaneira o desenvolvedor de tarefas morosas e pouco atraentes, e agilizando o ciclo de desenvolvimento como um todo.

CCM estende o modelo de objetos do CORBA, definindo funcionalidades e serviços que permitem que o desenvolvedor implemente, gerencie, configure e implante componentes que integrem os serviços do CORBA, como segurança, transações, persistência e eventos, em um ambiente padronizado.

CCM permite maior reutilização de servidores e mais flexibilidade para configuração dinâmica de aplicações CORBA.

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

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

2. O Modelo de Componentes

Component é um novo meta-tipo básico do CORBA, sendo uma extensão e especialização do meta-tipo *object*. Os componentes são especificados em IDL e podem ser representados no *Interface Repository*.

Um componente deve encapsular sua representação interna e implementação, sendo opaco ao cliente e exibindo em sua "superfície" funcionalidades com as quais os clientes, outros serviços do CORBA e elementos do ambiente irão interagir, denominadas portas (*ports*). As figuras 1.a e 1.b ilustram a definição em IDL de um componente e sua representação.

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

Figura 1.a: Representação em IDL de um componente Foo com seis interfaces, sendo duas disponíveis através de herança (A e B) e as demais por composição (W, X, Y e Z).

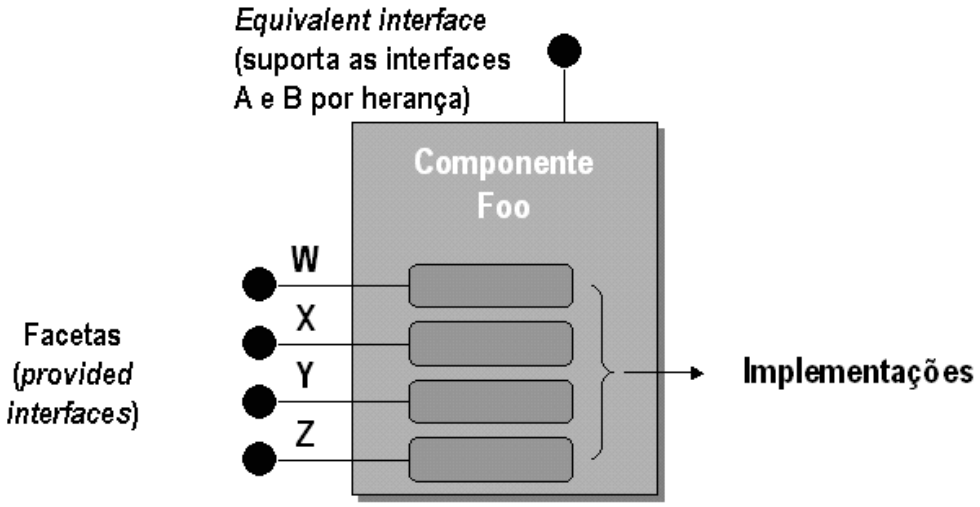

Figura 1.b: representação do componente da figura 1.a, ilustrando a opacidade da implementação das interfaces.

Uma referência a uma instância do componente Foo (das figuras 1.a e 1.b) aparece para os clientes como um objeto CORBA tradicional. Assim, clientes *component-unaware* podem invocar operações através de uma referência à *equivalent interface*, que é aquela que identifica unicamente a instância do componente, e que pode herdar de outras interfaces, chamadas *supported interfaces*.

Como anteriormente citado, é difícil estender objetos CORBA apenas por herança. Para resolver o caso, CCM define o conceito denominado facetas (*facets*), semelhante às interfaces dos componentes COM, permitindo suporte a interfaces não relacionadas entre si (*unrelated interfaces*), através de composição. Os clientes podem navegar pelas interfaces de um componente (facetas e *equivalent interfaces*) por intermédio da interface *Navigation*, definida em CORBA::CCMObject (herdada por todos os componentes), ou seja, a partir de uma referência a qualquer interface disponibilizada pelo componente é possível obter referência a qualquer de suas interfaces.

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

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

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

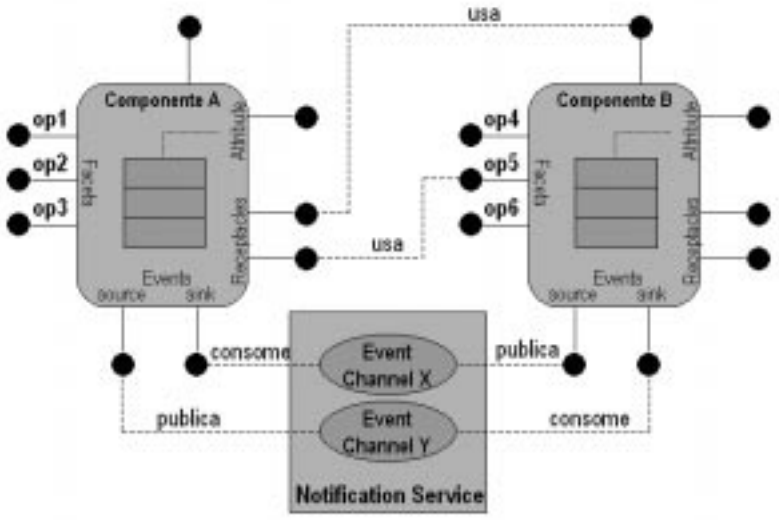

Figura 2: os tipos de portas do CCM.

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

- Atributos (*Attributes*): facilitam a configuração dos componentes, permitindo operações de acesso e atribuição;
- Chaves primárias (*primary keys*): valores expostos a clientes para identificar um determinado componente. Apenas os componentes que precisam ser localizados por operações *Finder*, chamados *keyed components*, possuem chaves primárias;
- Interface *Home*, que fornece operações padronizadas para *Factory* e *Finder* [GHJV94].

Utilização de componentes pelo cliente

O cliente deve obter uma referência à *home* do componente. Para isso, CCM define a interface *HomeFinder*, semelhante ao *OMG Naming Service* [OMG98], que implementa um serviço de diretório para *component homes*, e cuja utilização é:

resolve_initial_references(“HomeFinder”)

Uma vez com uma referência à *home* do componente, o cliente usa operações de *factory* ou *finder*, conforme apropriado, para obter uma referência ao componente em si.

Implementação de componentes

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

- Interface *Home*, vista anteriormente;
- Interface *Navigation*, que permite a obtenção de referência às facetas ou *equivalent interface*, através de uma referência válida a alguma interface do componente;
- Interfaces *Receptacles* e *Events;*

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

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

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

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

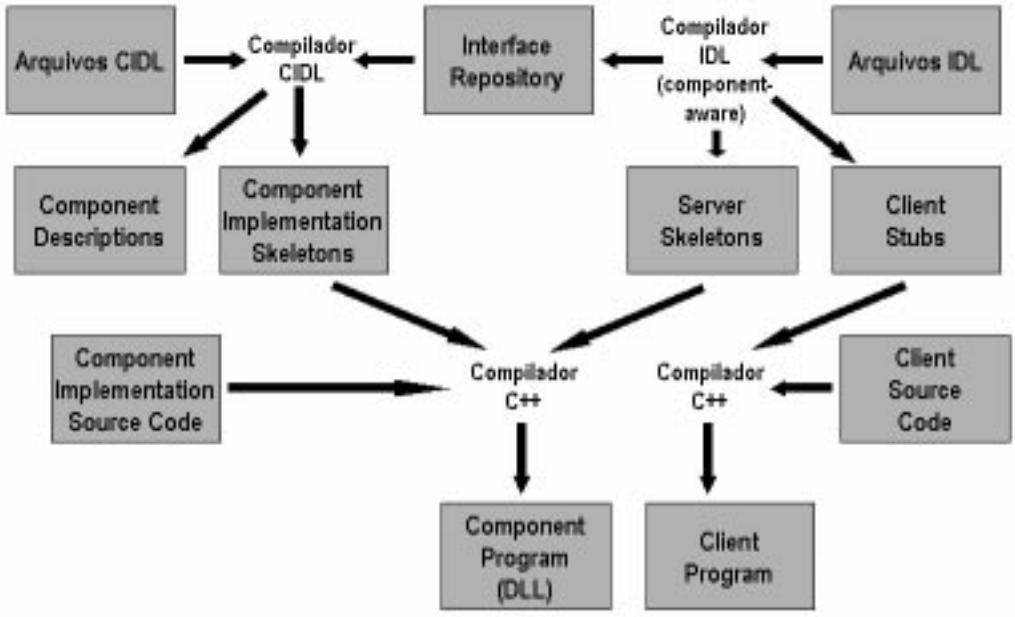

Figura 3: *Component Implementation Framework (CIF)*

3. A Arquitetura de Containers

Conforme citado anteriormente, os componentes são empacotados em DLLs e executados em *component servers*. As implementações dos componentes dependem do POA para encaminhar requisições de clientes para os serventes.

Os componentes não precisam saber como tratar problemas como a criação de hierarquia de POAs e localizar serviços do CCM. Para isso foram definidos os *containers*, que são a estrutura para integração dos serviços do CORBA, tais como transações, eventos, persistência e segurança, ao componente em tempo de execução, com as seguintes funcionalidades:

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

O gerenciamento do tempo de vida dos serventes é feito através de políticas que controlam o momento de ativação/desativação dos componentes:

- *Method*: ativação/desativação a cada chamada de método, limitando o uso de memória ao tempo de duração da operação, mas acrescentando o custo de ativação e desativação do componente;
- *Transaction*: ativação/desativação a cada transação. Memória permanece alocada durante a transação;
- *Component*: o *container* ativa o componente quando for feita a primeira chamada de alguma de suas operações, e desativa quando explicitamente requisitado pela aplicação, desalocando a memória utilizada pelo componente;
- *Container*: o componente será ativado quando for feita a primeira chamada a alguma de suas operações e, ao final da execução da mesma, será desativado. Entretanto, a memória permanecerá alocada até que o *container* decida desalocá-la.

Arquitetura do Modelo de Programação de *Containers*

Descreve a relação entre o *container* e os demais elementos do CCM. Fazem parte deste modelo de programação:

- Tipos externos, que correspondem às interfaces (incluindo a interface *Home*) disponíveis ao cliente, definidas em IDL e armazenadas no *Interface Repository*;
- Tipo do *container*, composto pelas interfaces internas e de *callback*, utilizadas pelo desenvolvedor do componente. O tipo do *container* é selecionado através de CIDL, sendo transiente ou persistente, dependendo do tipo de referências a objetos utilizado pelo componente;
- Tipo de implementação do *container*, que especifica o padrão da interação entre o *container*, o POA e os serviços do CORBA. Esses padrões são definidos em XML no descritor do componente, e usados pelo *factory* do *container* para criar o POA quando o *container* é criado, podendo ser: *stateless* – usa referências transientes a objetos com um

POA servente com suporte a qualquer ObjectId, *conversational* – usa referências transientes com POA servente dedicado a um ObjectId específico, e durável – usa referências persistentes com POA servente dedicado a um ObjectId específico;

- Categorias de componentes, que são o conjunto das combinações válidas entre os três elementos anteriores (conforme a Tabela 1), estendendo o *CORBA usage model* (que classifica referências a objetos como transientes ou persistentes), e são especificadas em arquivo CIDL.

Categoria de componente	Tipo de impl. do *container*	*CORBA Usage Model (Object Reference)*	*Keyed Home Interface*	Exemplo
Service	*Stateless*	Transiente	Não	*Wrappers* para aplic. procedurais de legado
Session	*Conversational*	Transiente	Não	*Iterators*
Process	Durável	Persistente	Não	*Regras de negócio*
Entity	Durável	Persistente	Sim	Peças num inventário

Tabela 1: relação entre categorias de componentes, tipo de implementação do *container* e tipo de referência a objetos

A arquitetura do modelo de programação de *containers* pode ser ilustrada pela Figura 4.

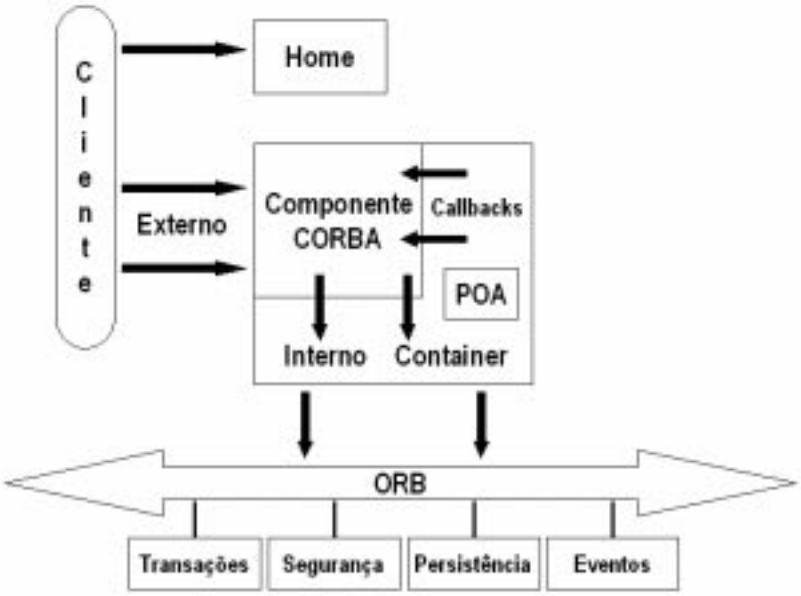

Figura 4: Arquitetura do modelo de programação de *containers*

4. Empacotamento e Implantação

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

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

Arquivo exemplo.idl: apresenta as definições dos componentes cliente e servidor, e uma interface para apresentação:

```
module exemplo
{
    // Interface para comunicação síncrona entre componentes.
    interface Apresenta
    {
         // Imprime o texto passado como parâmetro.
        void imprime(in string texto);
    };

    // Definição do componente cliente.
    component Cliente
    {
        // Propriedade que identifica a instância do componente.
        attribute string nome;

         // Receptáculo para conexão a um objeto Apresenta ou a uma faceta.
        uses Apresenta recep_servidor;
    };

    // Definição de Home para instanciar componentes clientes.
    // Não utilizado neste exemplo.
    home ClienteHome manages Cliente
    {
    };

    // Definição do componente servidor.
    component Servidor
    {
        // Propriedade que identifica a instância do componente.
        attribute string nome;

         // Faceta a ser utilizada pelo componente cliente.
        provides Apresenta faceta_para_clientes;
    };

    // Definição de Home para instanciar componentes servidores.
    // Não utilizado neste exemplo.
    home ServidorHome manages Servidor
    {
    };
};
```

Arquivo ServidorImpl.java: apresenta a implementação do componente exemplo::Servidor.

```
// Esta classe herda do esqueleto (classe) ServidorCCM,
// gerado pelo gerador de esqueletos IR3 (que faz parte do OpenCCM).
public class ServidorImpl
       extends ServidorCCM
       implements ApresentaOperations
{
    // Estado do componente
    private String nome_;
    private javax.swing.JTextArea janela_;

    // Construtor
    public ServerImpl()
    {
    }

    // Este método devolve uma referência a uma faceta, para ser
    // utilizada pelo componente cliente. Este método é requerido
    // pelo esqueleto ServidorCCM.
    public ApresentaOperations _get_facet_faceta_para_clientes()
    {
        // Devolve uma referência a si mesmo, pois implementa
         // a faceta Apresenta.
        return this;
    }

    // Método chamado para encerrar a fase de configuração do
    // componente, especificado em Components::CCMObject.
    public void configuration_complete ()
    throws org.omg.Components.InvalidConfiguration
    {
        // Testa se a configuração não está completa.
         if(nome_ == null)
             throw new org.omg.Components.InvalidConfiguration();

        // Instruções para inicializar a interface com o usuário (GUI).
         . . .
    }


    // Métodos para a interface exemplo::Servidor.

    // Método de atribuição (mutator) para o atributo nome.
    public void nome(String n)
    {
        nome_ = n;
    }

    // Método de acesso (accessor) ao atributo nome.
    public String nome()
    {
        return nome_;
    }


    // Métodos para a interface exemplo::Apresenta.

    // Método imprime.
    public void imprime(String texto)
```

```
    {
        // Apresenta o texto na área cliente da janela.
        janela_.append(texto + "\n");
    }
}
```

Arquivo ClienteImpl.java: apresenta a implementação do componente exemplo::Cliente.

```
// Esta classe herda do esqueleto ClienteCCM,
// gerado pelo gerador de esqueletos IR3.
// Esta classe também implementa java.awt.event.ActionListener,
// para tratamento das interações do usuário.
public class ClienteImpl
       extends ClienteCCM
       implements java.awt.event.ActionListener
{
    // Estado do componente.
    private String nome_;
    private javax.swing.JTextField texto_;

    // Construtor
    public ClienteImpl()
    {
    }

    // Método chamado para encerrar a fase de configuração do
    // componente, especificado em Components::CCMObject.
    public void configuration_complete ()
    throws org.omg.Components.InvalidConfiguration
    {
        // Testa se a configuração não está completa.
         if(nome_ == null)
            throw new org.omg.Components.InvalidConfiguration();

         // Verifica se a conexão com o servidor está estabelecida.
         if(get_connection_recep_servidor() == null)
            throw new org.omg.Components.InvalidConfiguration();

        // Instruções para inicializar a interface com o usuário (GUI).
         . . .
    }

    // Métodos para a interface exemplo::Cliente.

    // Método de atribuição (mutator) para o atributo nome.
    public void nome(String n)
    {
        nome_ = n;
    }

    // Método de acesso (accessor) ao atributo nome.
    public String nome()
    {
        return nome_;
    }

    // Método invocado quando o usuário pressionar um botão na tela.
    public void actionPerformed(java.awt.event.ActionEvent e)
```

```
    {
        // Obtém referência ao receptáculo recep_servidor.
        Apresenta recep_servidor = get_connection_recep_servidor();

        // Chama o método de impressão do servidor.
        recep_servidor.imprime(nome_ + ":" + texto_.getText());
    }
}
```

Arquivo exemplo.java: aqui os componentes são colocados em execução. Note a execução dos servidores de componentes, que serão os responsáveis pela criação dos *homes* para os componentes Servidor e Cliente, e a ligação do receptáculo de cada cliente à faceta do servidor.

```
public class exemplo
{
    // Procedimento inicial.
    public static void main(String[] args)
    throws Exception
    {
        // Inicializa o ORB.
        System.out.println("Initializando o ORB...");
         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);

        // Obtém referência ao serviço de nomes.
        System.out.println("Localizando o serviço de nomes...");
        org.omg.CORBA.Object obj =
            orb.resolve_initial_references("NameService");
        org.omg.CosNaming.NamingContext nc =
            org.omg.CosNaming.NamingContextHelper.narrow(obj);

        // Obtém referência a dois servidores de componentes:
        // um para criar home para "Servidor" e outro para "Cliente".
        System.out.println("Obtendo servidores de componentes...");
        org.omg.CosNaming.NameComponent[] ncomp = new
            org.omg.CosNaming.NameComponent[1];
        ncomp[0] = new
            org.omg.CosNaming.NameComponent("ServidorDeComponentes1", "");
        obj = nc.resolve(ncomp);
        OpenCCM.ComponentServer.Servidor servidor1 =
            OpenCCM.ComponentServer.ServerHelper.narrow(obj);
        ncomp[0].id = "ServidorDeComponentes2";
        obj = nc.resolve(ncomp);
        OpenCCM.ComponentServer.Servidor servidor2 =
            OpenCCM.ComponentServer.ServerHelper.narrow(obj);

        // Obtém os home factories e implantadores dos componentes.
        OpenCCM.ComponentServer.CCMHomeFactory fact_servidor1 =
            servidor1.provide_ccm_home_factory();
        org.omg.Components.Deployment.ComponentInstallation inst_servidor1 =
            servidor1.provide_install();
        OpenCCM.ComponentServer.CCMHomeFactory fact_servidor2 =
            servidor2.provide_ccm_home_factory();
        org.omg.Components.Deployment.ComponentInstallation inst_servidor2 =
            servidor2.provide_install();

        // Implanta os componentes. O arquivo exemplo.jar contém o conjunto dos
        // arquivos gerados de acordo com o Component Implementation Framework.
```

```
        System.out.println("Instalando arquivos...");
        inst_servidor1.install("exemplo", "./exemplo.jar");
        inst_servidor2.install("exemplo", "./exemplo.jar");

        // Cria os homes.
        System.out.println("Instanciando homes...");
        org.omg.Components.CCMHome h =
            fact_servidor1.create("OpenCCM.exemplo.ServidorHomeImpl", "sh");
        ServidorHome sh = ServidorHomeHelper.narrow(h);
        h = fact_servidor2.create("OpenCCM.exemplo.ClienteHomeImpl", "ch");
        ClienteHome ch = ClienteHomeHelper.narrow(h);

        // Cria os componentes.
        System.out.println("Instanciando componentes...");
        Servidor s = sh.create();
        Cliente c1 = ch.create();
        Cliente c2 = ch.create();
        Cliente c3 = ch.create();

         // Configura os componentes.
        System.out.println("Configurando componentes...");
        s.name("Meu Servidor");
        c1.name("Cliente 1");
        c2.name("Cliente 2");
        c3.name("Cliente 3");

        // Conecta os receptáculos dos clientes
         // à faceta do servidor.
        System.out.println("Conectando componentes...");
        Apresenta faceta_para_clientes = s.provide_faceta_para_clientes();
        c1.connect_recep_servidor(faceta_para_clientes);
        c2.connect_recep_servidor(faceta_para_clientes);
        c3.connect_recep_servidor(faceta_para_clientes);

        // Indica final da fase de configuração.
        System.out.println("Configuração concluída...");
        s.configuration_complete();
        c1.configuration_complete();
        c2.configuration_complete();
        c3.configuration_complete();

       // Abandona o programa principal.
       System.exit(0);
    }
}
```

6. Conclusão

Até o presente momento, a especificação do CORBA 3 já foi amplamente discutida, e está prestes a ser lançada no mercado. Entretanto, os principais fornecedores de ORBs, como IONA, Inprise e BEA Systems já estão implementando as novas funcionalidades, sendo que em breve teremos alguns produtos comerciais. Para aqueles que desejam realizar testes, existem pelo menos duas implementações gratuitas do CCM, em versão beta, disponíveis para obtenção via Internet:

- OpenCCM, escrita em Java [GOAL01];

- Implementação para MICO, escrita em C++ [Ruiz01];

Ainda fazendo parte do CCM, há um capítulo da especificação destinado à integração entre CCM e *Enterprise Java Beans* (EJB) [Thom98], significando que um componente CCM poderá ser visto como um componente EJB e vice-versa, oferecendo flexibilidade e reaproveitamento do trabalho já realizado, uma vez que será possível, por exemplo, aproveitar a elegância da linguagem Java (de um componente EJB já existente) com a eficiência do C++ (em componentes CCM).

Referências

[Box98] Don Box. *Essential COM*. Addison-Wesley, 1998

[CCM99] Object Management Group. *CORBA Component Model Joint Revised Submission*. 1999

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994

[GOAL01] Genie des Objets et composAnts Logiciels (GOAL) Team,
Laboratoire d'Informatique Fondamentale de Lille (LIFL),
Universite des Sciences et Technologies de Lille (USTL). *Open CORBA Component Model Platform*. 2000-2001

[HV99] M. Henning, S. Vinoski. *Advanced CORBA Programming with C++.* Addison-Wesley, 1999

[OMG98] Object Management Group. *Interoperable Naming Service Specification*. 1998

[OMG99] Object Management Group. *The Common Object Request Broker: Architecture and Specification, 2.3 ed*. 1999

[Prit99] Jason Pritchard. *COM and CORBA Side by Side.* Addison-Wesley, 1999)

[Ruiz01] Diego Sevilla Ruiz. *CORBA & Component Model Web Site - http://www.ditec.um.es/~dsevilla/ccm/*

[Thom98] A. Thomas, Patricia Seybold Group. *Enterprise JavaBeans Technology – http://java.sun.com/products/ejb/white_paper.html*. 1998

[WLS00] N. Wang, D. Levine, D. Schmidt. *Optimizing the CORBA Component Model for High-performance and Real-time*