# SCS - Sistema de Componentes de Software

C. E. L. Augusto, E. Fonseca, L. Marques, H. Roenick
R. F. G. Cerqueira, S. L. Corrêa
Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)
rcerq@inf.puc-rio.br

**Resumo**

Este documento descreve um sistema de componentes de software para a arquitetura CORBA, denominado SCS (Sistema de Componentes de Software), cujo objetivo é prover uma infra-estrutura leve e simples para distribuir, instanciar e executar aplicações baseadas em componentes de software.

## 1 Introdução

A construção de aplicações distribuídas tem sido grandemente apoiada pelo uso de *middlewares* baseados em objetos distribuídos como, por exemplo, CORBA e RMI. Estes, fornecem uma camada de software entre as aplicações e o sistema operacional, fornecendo os serviços comuns de distribuição, heterogeneidade e portabilidade. Entretanto, *middlewares* baseados em objetos distribuídos enfrentam algumas dificuldades, as quais tendem a se agravar à medida que os sistemas se tornam mais complexos [1]. Dentre tais dificuldades, a falta de uma abstração para agrupar interfaces relacionadas em uma família de serviços e a ausência de um mecanismo de implantação e configuração dos objetos de uma aplicação têm sido um grande problema para aplicações construídas a partir destas infra-estruturas. A ausência de abstrações mais complexas tem produzido aplicações formadas por um grande número de objetos, a maioria dos quais representando unidades simples e com poucas responsabilidades. Como consequência, sistemas construídos a partir destes *middlewares* se tornam aplicações grandes, complexas e difíceis de serem gerenciadas. De maneira análoga, a ausência de uma infra-estrutura para distribuir, instalar, instanciar e ativar implementações de objetos obriga que as aplicações implementem seus próprios mecanismos de forma não padronizada, dificultando a portabilidade das mesmas.

Uma abordagem que tem se mostrado eficiente para resolver as limitações de *middlewares* baseados em objetos distribuídos consiste no uso de *middlewares* baseados em componentes de software [2]. Estes se caracterizam por criar a noção de fronteira em volta das implementações dos componentes de uma aplicação, os quais se comunicam apenas através de interfaces bem definidas. Estes componentes constituem-se em unidades de construção de sistemas com funcionalidades e responsabilidades claramente estabelecidas, podendo também apresentar mais de uma visão (ou papéis) para seus clientes. Dessa forma, componentes de software são abstrações mais complexas que objetos.

A proposta de componentes de software ressurgiu recentemente como uma forma de complementar o modelo orientado a objetos e prover meios de diminuir suas deficiências. Dessa forma, o modelo de componentes de software incorpora vários conceitos do paradigma de objetos, como encapsulamento e separação entre interface e implementação, mas além disso também torna explícitos conceitos como dependências e conexões entre componentes.

Componentes de software podem ser definidos como unidades de composição com interfaces contratualmente especificadas e dependências de contexto explícitas, além de poderem ser independentemente implantados e estar sujeitos a composição por terceiros [3]. Tal separação entre a definição do componente (i.e. suas interfaces, serviços e dependências) e sua implementação, permite que o mesmo possa ser manipulado como caixa-preta, ou seja, com base exclusivamente na sua definição. Esta, por sua vez, especifica um conjunto de conectores através dos quais é possível acessar os serviços do componente, bem como fornecer os recursos por ele esperados, definidos como suas dependências.

A construção de aplicações baseado em componentes é feita, então, estabelecendo-se conexões entre componentes de software através da ligação de seus conectores, de forma que as dependências de um componente sejam supridas pelos serviços oferecidos por outro. Neste contexto, um *middleware* baseado em componentes envolve um modelo que define a estrutura dos componentes do sistema, um modelo que descreve a forma de interação entre eles e a definição de um ambiente genérico para execução destes componentes.

Este documento descreve um *middleware* baseado em componentes para a arquitetura CORBA, denominado SCS (Sistema de Componentes de Software). Inicialmente, na seção 2, apesentamos uma visão geral do SCS, descrevendo seu modelos de componentes (estrutura, conectores e dependências) e seu ambiente de execução (mecanismo de instanciação e execução dos componentes de uma aplicação). Posteriormente, na seção 3, discutimos alguns sistemas de componentes existentes e relacionados ao SCS, justificando nossa opção por este último. Finalmente, descrevemos algumas conclusões (seção 4).

# 2 Visão Geral

## 2.1 Modelo de Componentes

SCS é um sistema de componentes de software projetado para a arquitetura CORBA cujo objetivo é prover uma infra-estrutura de distribuição, instalação, configuração e execução de componentes. Inspirado em COM cite e CCM cite, seu modelo de componentes foi idealizado visando flexibilidade, simplicidade e facilidade de uso através de um conjunto pequeno de APIs, as quais são implementadas pelos componentes de acordo com suas necessidades. Um componente em SCS é uma unidade lógica pronta para composição e reuso, podendo também encapsular um modelo de interação, configuração e introspecção. A seguir descrevemos cada modelo.

### 2.1.1 Modelo de Interação

Como mencionado anteriormente, um sistema de componentes constrói aplicações por composição através de portas de serviços que se conectam entre si. Portas de serviço definem os conectores através dos quais um componente pode ser interligado a outros componentes para formar um sistema. Neste sentido, SCS provê dois tipos de portas de serviços: facetas (ou portas de provisão de serviços) e receptáculos(ou portas de requisição de serviços).

## Facetas

Facetas são portas de provisão de serviços que oferecem uma determinada interface. Comumente, os serviços oferecidos pelo componente são disponibilizados através de um conjunto de facetas. Adicionalmente, é possível que um componente ofereça uma mesma interface através de portas distintas, mas com diferentes implementações e comportamentos. Facetas são implementadas por objetos que expõem interfaces CORBA comuns, podendo ser acessadas por clientes CORBA quaisquer.

Em SCS, uma faceta é definida por um tipo, um nome simbólico atribuído pelo desenvolvedor e o objeto CORBA que a implementa. O tipo de uma faceta, por sua vez, consiste na interface correspondente, a qual é definida em IDL (*Interface Definition Language*). Uma vez definida uma faceta, a mesma pode ser identificada através do seu tipo ou do seu nome simbólico.

## Receptáculos

Receptáculo são portas através das quais um componente requisita um serviços, ou seja, são pontos de acesso a serviços os quais um componente depende ou utiliza. Co-

nexões entre componentes são estabelecidas conectando-se a faceta de um componente provedor ao receptáculo de um componente requisitante do serviço.

É importante mencionar que o modelo de componentes definido para o SCS não impõe nenhuma restrição quanto à cardinalidade das conexões, ou seja, é possível que uma faceta seja conectada a nenhum ou a vários receptáculos simultaneamente. Analogamente, um receptáculo pode estar conectado a diversas implementações de objetos de uma mesma interface, situação em que o mesmo é denominado receptáculo múltiplo. Em SCS, um receptáculo é definido por um nome simbólico atribuído pelo desenvolvedor, o nome da interface que se conecta ao receptáculo, os objetos CORBA que a implementam e que estão conectados ao receptáculo e um *flag* para indicar se o receptáculo é múltiplo ou simples. Entretanto, uma vez definido, um receptáculo é identificado unicamente pelo seu nome simbólico.

### 2.1.2 Modelo de Configuração

O modelo de interação propicia, de forma inerente, um modelo de configuração com alta granularidade. Através de operações de conexões, componentes inteiros podem ser substituídos em tempo de execução, mudando-se a configuração das aplicações. Adicionalmente, o conceito de facetas permite que novas interfaces sejam adicionadas estaticamente a componentes já existentes, sem contudo, afetar as aplicações que já utilizam seus serviços. Esta capacidade é especialmente interessante para o gerenciamento de versões, uma vez que é possível evoluir ou disponibilizar novas versões de um serviço sem contudo modificar os clientes de versões anteriores.

### 2.1.3 Modelo de Introspecção

O modelo de componentes de SCS define um conjunto de operações que permitem inspecionar, em tempo de execução, um componente de diversas formas. Em geral, é possível examinar as facetas oferecidas por um componente, bem como as conexões por ele mantidas.

### 2.1.4 Componente em SCS

A figura 1 mostra a estrutura de um componente SCS. Em geral, um componente pode apresentar quantidades quaisquer de facetas e receptáculos. Porém, três interface específicas são oferecidas pelo modelo e podem compor o comportamento de um componente SCS:

- `IComponent`, a qual define o tipo "componente"em SCS e que define as operações para ativação e desativação de um componente, bem como operações para requisição de uma faceta.

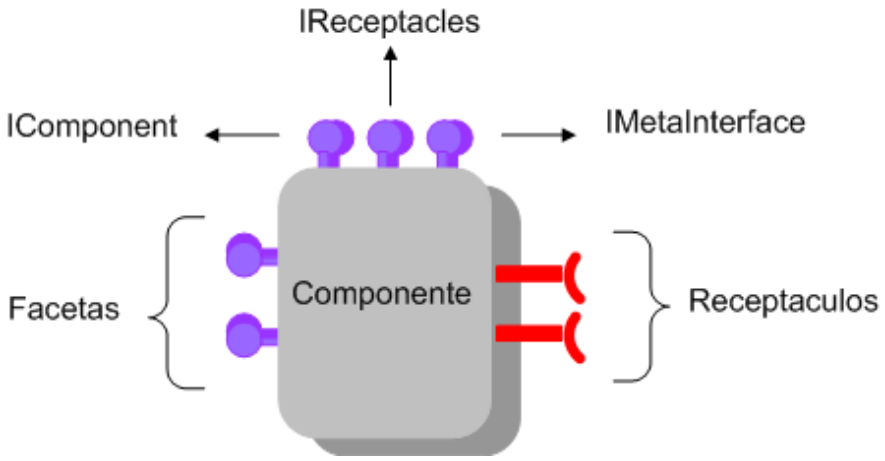


Figura 1: Um componente em SCS

- `IReceptacles`, que define operações para gerenciar conexões de receptáculos, como por exemplo, métodos para conectar e desconectar um objeto a um receptáculo e métodos para listar todos os objetos a ele conectados.
- `IMetaInterface`, que define operações básicas para introspecção de facetas e receptáculos do componente.

Todo componente deve implementar pelo menos uma interface, a `IComponent`, sendo as interfaces `IReceptacles` e `IMetaInterface` opcionais. A API de cada interface é descrita a seguir. Antes porém, é importante descrever as estruturas que definem as portas SCS e que são utilizadas nas assinaturas dos métodos destas interfaces.

No listing 1, são definidas as estruturas `FacetDescription` (linhas 1-5) e `ReceptacleDescription` (linhas 14-19) as quais definem, respectivamente, uma faceta e um receptáculo. A estrutura `FacetDescription` é composta pelos campos `name`, `interface_name` e `facet_ref`, os quais representam, respectivamente, o nome simbólico da faceta, o nome da interface que identifica seu tipo, e o objeto CORBA que a implementa. A estrutura `ReceptacleDescription` é composta pelos campos `name`, `interface_name`, `is_multiplex` e `connections`, que representam, respectivamente, o nome simbólico do receptáculo, o nome da interface que se conecta a ele, o tipo do receptáculo (simples ou múltiplo) e suas conexões. Cada conexão (estrutura `ConnectionDescription`, linhas 8-11), por sua vez, é definida por um identificador (`id`) e o objeto CORBA conectado (`objref`). As sequências `FacetDescriptions` (linha 6), `ConnectionDescriptions` (linha 12) e `ReceptacleDescriptions` (linha 20) representam listas de facetas, conexões e receptáculos respectivamente.

Listing 1: Estruturas que definem uma faceta e um receptáculo

```
1     struct FacetDescription {
2          string name;
3          string interface_name;
4          Object facet_ref;
5      };
6      typedef sequence<FacetDescription> FacetDescriptions;
7
8      struct ConnectionDescription {
9      ConnectionId id;
10     Object objref;
11   };
12   typedef sequence<ConnectionDescription> ConnectionDescriptions;
13
14     struct ReceptacleDescription {
15     string name;
16     string interface_name;
17     boolean is_multiplex;
18     ConnectionDescriptions connections;
19   };
20   typedef sequence<ReceptacleDescription> ReceptacleDescriptions;
```

### *IComponent*

Como ilustrado no listing 2, a interface `IComponent`, linhas 7-14, define as seguintes operações:

- `startup`: este método é invocado para ativar o componente. Ao ser chamado, o componente já deve estar devidamente inicializado e com suas dependências externas resolvidas.
- `shutdown`: este método é invocado para desativar o componente. Após sua chamada, o componente não deve mais estar apto para receber chamadas remotas.
- `getFacet`: este método retorna a faceta de um componente, dada a interface que ela implementa.
- `getFacetByName`: este método retorna a faceta de um componente, dado seu nome simbólico.
- `getComponentId`: este método retorna um `ComponentId` (definida nas linhas 1-4). Esta estrutura identifica uma implementação de um componente, através de um nome simbólico (campo `name`) e e a versão da implementação (campo `version`).

***IReceptacles***

Como ilustrado no listing 2, a interface `IReceptacles`, linhas 16-24, define as seguintes operações:

- `connect`: este método é invocado para conectar um receptáculo do componente (identificado pelo parâmetro `receptacle`) a um objeto CORBA (identificado pelo parâmetro `obj`), retornando para o invocador uma identificação para a conexão criada.
- `disconnect`: este método é invocado para desfazer a conexão de um receptáculo a um objeto. Para tanto, a conexão a ser desfeita é passada como parâmetro para o método.
- `getConnections`: este método retorna todas as conexões ativas de um receptáculo, fornecido como parâmetro para o método.

***IMetaInterface***

Como ilustrado no listing 2, a interface `IMetaInterface`, linhas 26-33, define as seguintes operações:

- `getFacets`: retorna todas as facetas do componente.
- `getFacetsByName`: retorna uma lista de facetas, dado uma lista contendo seus nomes simbólicos.
- `getReceptacles`: retorna todos os receptáculos do componente.
- `getReceptaclesByName`: retorna uma lista de receptáculos, dado uma lista contendo seus nomes simbólicos.

## 2.2 Ambiente de Execução

O Ambiente de Execução em SCS consiste em um modelo que define um mecanismo padrão para instanciação, carga, configuração e execução dos componentes do sistema. Dois serviços principais formam tal modelo: contêiner e nó de execução.

### 2.2.1 Contêiner

Um contêiner é uma abstração que define um ambiente de execução protegido onde implementações de componentes são implantadas, criadas e executadas. Interações entre a implementação de um componente e o mundo externo são intermediadas pelo

Listing 2: Interfaces que definem um componente SCS

```
1  struct ComponentId {
2      string name;
3      unsigned long version;
4  };
5  typedef sequence<ComponentId> ComponentIdSeq;
6
7  interface IComponent {
8      void startup() raises (StartupFailed);
9      void shutdown() raises (ShutdownFailed);
10
11     Object getFacet (in string facet_interface);
12     Object getFacetByName (in string facet);
13     ComponentId getComponentId ();
14 };
15
16 interface IReceptacles {
17     ConnectionId connect (in string receptacle, in Object obj)
18        raises (InvalidName, InvalidConnection, AlreadyConnected,
19                ExceededConnectionLimit);
20     void disconnect (in ConnectionId id)
21        raises (InvalidConnection, NoConnection);
22     ConnectionDescriptions getConnections (in string receptacle)
23        raises (InvalidName);
24 };
25
26 interface IMetaInterface {
27     FacetDescriptions getFacets();
28     FacetDescriptions getFacetsByName(in NameList names)
29        raises (InvalidName);
30     ReceptacleDescriptions getReceptacles();
31     ReceptacleDescriptions getReceptaclesByName(in NameList names)
32        raises (InvalidName);
33 };
```

contêiner, o qual oferece serviços e funcionalidades às implementações de componentes, minimizando, dessa forma, o esforço de implementação por parte dos mesmos. Isto promove também uma forma padronizada para implantar, criar, ativar e fornecer serviços a implementações de componentes. De maneira geral, podemos listar as seguintes responsabilidades para um contêiner:

- Criação de componentes: Um contêiner deve ser capaz de construir uma instância inteiramente funcional de um componente, a partir da sua implementação.
- Ativação de componentes: Um contêiner deve ser capaz de ativar e desativar um componente de acordo com determinadas políticas, bem como controlar o estado das conexões do componente.
- Definição de políticas para controle de qualidade de serviços: Contêiners diferentes podem ser definidos para tratar ou garantir diferentes tipos de qualidade de serviços, como tolerância a falhas ou reserva de recursos.

- Acesso a serviços CORBA: Um contêiner deve prover acesso aos serviços de objetos CORBA (como eventos, transações, persistência, etc.), fornecendo assim um mecanismo padrão de obtenção desses serviços, bem como compartilhamento dos mesmos visando economia de recursos.

Em SCS, contêiners são implementados através de componentes denominados `Container`, figura 2. Tais componentes definem um espaço de endereçamento para os componentes de uma aplicação, isolando-as dos componentes de outras aplicações. Em geral, componentes de uma aplicação SCS executam no mesmo espaço de endereçamento do seu componente `Container`, o qual possui dois tipos de facetas:

- `ComponentLoader`, que oferece as operações para o carregamento de componentes.
- `ComponentCollection`, que permite consultar os componentes carregados no contêiner.

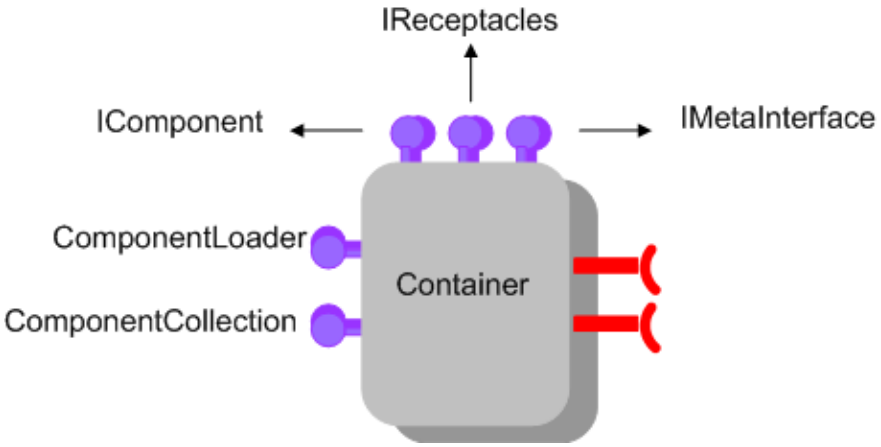


Figura 2: Componente *Container*

É importante mencionar também que um contêiner SCS gerencia implementações de componentes em uma mesma linguagem de programação. Dessa forma, todos os componentes em um contêiner são implementados em uma mesma linguagem. Essa decisão de projeto visa garantir simplicidade no uso de um contêiner SCS, visto que contêiners multi-linguagens são utilizados raramente e não justificam sua complexidade adicional.

O listing 3 mostra as interfaces implementadas por um componente `Container`. As linhas 1 a 5 mostram a definição da estrutura `ComponentHandle` usada para identificar uma instância de um componente. O campo `cmp` é a instância do componente; o campo `id` identifica a implementação do componente; o campo `instance_id`

é uma chave de identificação da instância. A seguir descrevemos as interfaces correspondentes às facetas de um componente `Container`.

***ComponentLoader***

Como ilustrado no listing 3, a interface `ComponentLoader` (linhas 8-14) implementa as seguintes operações:

- `load`: Esta operação carrega (ou instancia) um componente com a implementação identificada pelo parâmetro do tipo `ComponentId` e retorna um descritor do tipo `ComponentHandle`, para identificar a instância carregada. O parâmetro `args` é um vetor de strings que deve ser passado como parâmetro para o ponto de entrada do componente.
- `unload`: Esta operação descarrega a instância do componente identificada pelo parâmetro do tipo `ComponentHandle`.
- `getInstalledComponents`: Esta operação retorna os identificadores de todas as implementações de componentes conhecidas pelo contêiner, e passíveis de instanciação.

***ComponentCollection***

Como ilustrado no listing 3, a interface `ComponentCollection` (linhas 16-19) implementa as seguintes operações:

- `getComponent`: Esta operação retorna uma seqüência com todas as instâncias de uma determinada implementação de componente (`ComponentId`) carregadas no contêiner.
- `getComponents`: Esta operação retorna uma seqüência com todas as instâncias de componentes carregadas em um contêiner.

### 2.2.2 Nós de Execução

Num ambiente de execução, um nó de execução corresponde a um dispositivo físico (*host*) onde contêiners executam. Em SCS um nó de execução é representado por um componente denominado `ExecutionNode` (figura 3), cuja finalidade é gerenciar os contêiners nele instanciados. Para tanto, este componente implementa uma faceta, também denominada `ExecutionNode`, a qual oferece operações para disparar novos contêiners e consultar os contêiners em execução. A seguir definimos tal interface com mais detalhes.

Listing 3: Interfaces que definem um componente `Container`

```
1  struct ComponentHandle {
2       core::IComponent cmp;
3       core::ComponentId id;
4       unsigned long instance_id;
5  };
6  typedef sequence<ComponentHandle> ComponentHandleSeq;
7
8  interface ComponentLoader {
9       ComponentHandle load (in core::ComponentId id, in StringSeq args)
10         raises (ComponentNotFound, ComponentAlreadyLoaded, LoadFailure);
11      void unload (in ComponentHandle handle)
12         raises (ComponentNotFound);
13      core::ComponentIdSeq getInstalledComponents();
14 };
15
16 interface ComponentCollection {
17   ComponentHandleSeq getComponent (in core::ComponentId id);
18   ComponentHandleSeq getComponents ();
19 };
```

### `ExecutionNode`

Como ilustrado no listing 4, a interface `ExecutionNode` (linhas 15-23) define:

- um atributo `name`: Este atributo representa o nome simbólico que identifica o nó de execução. Tipicamente, seu valor consiste no nome da máquina ou *host*.
- `startContainer`: Esta operação dispara um novo contêiner. O parâmetro `container_name` é o nome simbólico que deve ser associado ao novo contêiner. O parâmetro `props` (do tipo `Property`, definido nas linhas 1- 5) pode definir um conjunto de propriedades que caracterizem o contêiner que deve ser disparado. Exemplos de tais propriedades consistem em linguagem de programação do contêiner (C++, Lua ou Java), eventuais argumentos que devem ser passados para o mesmo durante a sua inicialização, etc.
- `getContainer`: Esta operação retorna o contêiner identificado pelo parâmetro `container_name`.
- `getContainers`: Esta operação retorna uma seqüência com todos os contêiners registrados no nó de execução. Cada contêiner retornado é descrito pela estrutura `ContainerDescription` (linhas 8- 12), onde o campo `container` é uma referência para o componente `Container`, o campo `container_name` é um nome simbólico usado para identificá-lo e o campo `execution_node` é uma referência para o nó de execução que o hospeda.

A interface `ContainerManager` mostrada no listing 4, linhas 25-29, é usada pelo Execution Node para controlar a criação do container. Esta interface define o

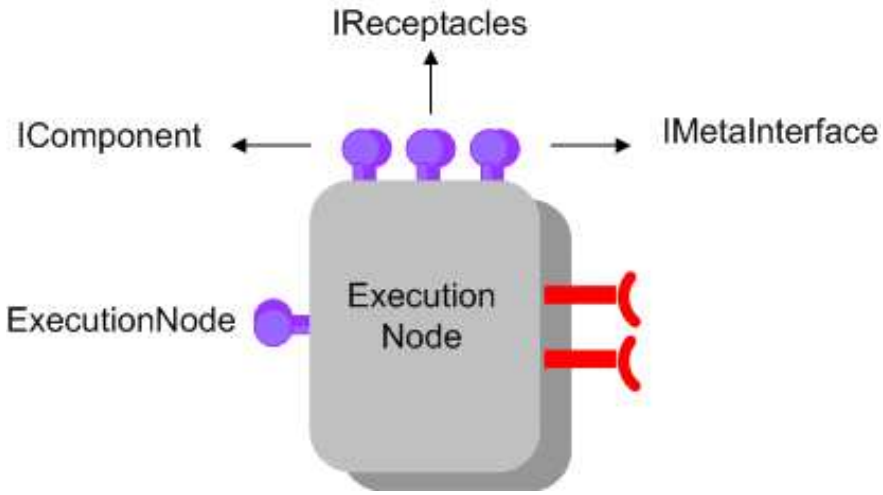


Figura 3: Componente `ExecutionNode`

método `registerContainer`, no qual o contêiner recem-criado deve se registrar para indicar ao Execution Node que sua criação foi bem sucedida. Adicionalmente, o método `unregisterContainer` é oferecido para deregistrar um contêiner.

# 3 Tecnologias Relacionadas

Como mencionado anteriormente, SCS foi inspirado nos modelos de componentes de COM ( Component Object Model) [4] e CCM (CORBA Component Model) [1]. COM consiste em uma tecnologia de componentes da Microsoft, tendo sido o primeiro modelo de componentes de software a alcançar grande popularidade. Entretanto, diferentemente de SCS e CCM, COM foi projetado inicialmente para suportar um modelo de programação de componentes não distribuído, sendo a distribuição adicionada apenas posteriormente com a tecnologia DCOM. Também, diferentemente de SCS e CCM, que utilizam a tecnologia CORBA, COM está limitado às plataformas Microsoft. CCM é um modelo de componentes proposto pela OMG, cujo objetivo é estender o modelo de objetos CORBA, com o intuito de tratar alguns problemas deixados em aberto pela arquitetura. Inspirado no modelo de componentes de EJB (Enterprise Java Beans), CCM é considerado um dos modelos comerciais mais completos atualmente, oferecendo serviços refinados de configuração e deployment de componentes. Por isto mesmo, a especificação CCM é grande e complexa. Quando comparado ao modelo de componentes SCS, podemos afirmar que tanto SCS quanto CCM permitem a interoperabilidade entre diferentes plataformas e linguagens de programação, uma vez que ambos modelos de componentes se baseiam na arquitetura CORBA. Entretanto, diferentemente de CCM, SCS se propõe a ser um modelo simples e leve, enfatizando, dessa

Listing 4: Interface que define um componente `ExecutionNode`

```
1  struct Property {
2       string name;
3       string value;
4       boolean read_only;
5  };
6  typedef sequence<Property> PropertySeq;
7
8  struct ContainerDescription {
9       core::IComponent container;
10      string container_name;
11      core::IComponent execution_node;
12 };
13 typedef sequence<ContainerDescription> ContainerDescriptionSeq;
14
15 interface ExecutionNode {
16      core::IComponent startContainer (in string container_name, in PropertySeq props)
17         raises (ContainerAlreadyExists, InvalidProperty);
18      void stopContainer(in string container_name)
19         raises (core::InvalidName);
20      core::IComponent getContainer (in string container_name);
21      ContainerDescriptionSeq getContainers ();
22      string getName();
23 };
24
25 interface ContainerManager {
26         void registerContainer(in string name, in core::IComponent ctr)
27           raises (ContainerAlreadyExists, InvalidContainer);
28          void unregisterContainer(in string name) raises (core::InvalidName);
29   };
```

forma, a facilidade de utilização. Outro modelo de componentes de grande aceitação consiste no EJB [5]. Como CCM, este modelo apresenta uma especificação complexa, oferecendo serviços refinados de configuração e deployment de componentes. Entretanto, diferentemente de CCM, EJB está limitado à tecnologia Java.

# 4 Conclusão

Middlewares baseados em objetos distribuídos apresentam dificuldades para desenvolver e gerenciar aplicações grandes e complexas. Como conseqüência, o modelo de componentes de software tem sido usado como uma extensão domodelo de objetos, visando suprir suas deficiências. Neste sentido, middlewares baseados em componentes de software se mostram mais eficientes para atender as necessidades das novas aplicações, uma vez que eles oferecem abstrações mais complexas e uma infra-estrutura padronizada para gerenciamento e deployment de seus componentes. SCS é um modelo de componentes de software baseado na arquitetura de comunicação de objetos distribuídos CORBA, que permite a interoperabilidade entre diferentes plataformas e linguagens de programação. Embora outrosmodelos de componentes para CORBA

já existam, eles tendem a ser grandes e complexos, o que desencoraja seu uso. SCS, ao contrário, se propõe a oferecer um serviço básico de deployment de componentes, porém simples e fácil de ser usado.

# Referências


[1] N. Wang, D. C. Schmidt, and C. O'Ryan, *An Overview of the CORBA Component Model. COMPONENT-BASED SOFTWARE ENGINEERING*. Addison-Wesley, 2000.

[2] A. Gokhale, D. C. Schmidt, B. Natarajan, and N. Wang, "Applying model-integrated computing to component middleware and enterprise applications," *Communications of the ACM*, vol. 45, no. 10, 2002.

[3] C. Szyperski, *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, 2002.

[4] D. Box, *Essential COM.* Addison-Wesley, 1997.

[5] A. Thoma, "Enterprise java beans technology," 1998. Available on: `http://java.sun.com/products/ejb/white_paper.html`.