Tutorial Básico do SCS C#

Tecgraf
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)
scs-users@tecgraf.puc-rio.br

2011-08-31

1 Introdução

Este documento é um tutorial básico sobre a criação de componentes no modelo SCS v1.2[4]. Não serão encontradas aqui explicações sobre o modelo, as quais encontram-se em documentos específicos. Também não será abordado o uso de serviços específicos desenvolvidos para o auxílio ao uso do modelo, como a infra-estrutura de execução. Essas informações também podem ser obtidas em outros documentos. A implementação do SCS C# versão 1.2.1_3 utiliza o .Net Framework 4.0[3] e IIOP.NET 1.9.4.0[2]. Este documento assume que o leitor é familiarizado a conceitos de desenvolvimento de *software* baseado em componentes e à terminologia CORBA[1].

2 Inicialização do ORB e Carga da IDL do SCS

Para a criação e execução do código de um componente, é necessária a inicialização prévia do ORB. O IIOP.NET não implementa CORBA completamente, para inicializar um ORB é necessário seguir os passos do Código 1.

A classe OrbServices possui três métodos estáticos para a criação e registro de canais de comunicação IIOP. Para criar um canal apenas cliente é utilizado o método

Código 1: Criação do ORB

```
1  int port = ...
2  IiopChannel chan = OrbServices.CreateAndRegisterIiopChannel(port);
```

com assinatura sem parâmetros. Já para criar um canal que aceite receber requisições, é necessário utilizar o método que possui a porta do canal como parâmetro, ou o método que recebe um mapa de propriedades, definindo a propriedade referente à porta e/ou outras especificadas pela documentação do IIOP.NET.

3 Passos Necessários à Criação de um Componente

Serão descritos os passos mínimos necessários para a criação de um componente SCS.

3.1 Definição do Identificador do Componente

O identificador do componente é uma estrutura definida em IDL (scs.idl) chamada ComponentId e é representada por uma *struct* que possui os seguintes campos:

- `name`: Nome desejado para o componente.
- `major_version`: Número que define a versão principal do componente.
- `minor_version`: Número que define a versão secundária do componente, possivelmente relacionado a uma sub-versão da versão principal.
- `patch_version`: Número que define a versão de revisão do componente.
- `platform_spec`: *String* contendo quaisquer especificações de plataforma necessárias ao funcionamento do componente.

Os números de versão do componente, quando unificados, devem ser separados por pontos. Ou seja, um componente com versão principal 1, versão secundária 0 e versão de revisão 0 deve ser representado como a *String* "1.0.0".

3.2 Criação do Componente Básico

O componente SCS é representado pela interface *ComponentContext*. A classe *DefaultComponentContext* é uma implementação padrão da interface, ela atua como um invólucro local para as facetas e receptáculos de um componente SCS.

A classe *DefaultComponentContext* está localizada em *Scs.Core* e seu processo de instanciação engloba a criação das três facetas básicas, *IComponent*, *IReceptacles* e *IMetaInterface*. Caso o usuário necessite utilizar uma implementação diferente de alguma dessas facetas, basta utilizar o método *UpdateFacet*.

Um exemplo de código para a criação de um componente básico pode ser visto no Código 2.

Código 2: Instanciação de um Novo Componente

```
1 ComponentId componentId = new ComponentId("MyComponent", 1, 0, 0, ".NET Framework");
2 ComponentContext context = new DefaultComponentContext(componentId);
```

3.3 Criação de Facetas

Facetas são interfaces CORBA, e devem ser implementadas pelo desenvolvedor da aplicação. O IIOP.NET obriga que as facetas implementadas tenham modificador de acesso *public*, estenda MarshalByRefObject e implemente a interface da faceta.

Um exemplo de implementação de faceta pode ser visto no Código 3. Essa faceta precisa ter uma especificação em IDL. Para o nosso exemplo, utilizaremos a IDL contida no Código 4.

Código 3: Implementação de uma Faceta MyFacet

```
1 public class MyFacetServant : MarshalByRefObject, MyFacet {
2   MyFacetServant(ComponentContext context) { this.context = context }
3
4   public void myMethod() {
5     ...
6   }
7 }
```

Um detalhe importante é que, devido à forma como objetos do tipo "MarshalByRefObject"podem definir sua política de "tempo de vida", os *servants* podem ser co-

Código 4: Exemplo de IDL de uma Faceta

```
module mymodule{
  interface MyFacet {
    void myMethod();
  };
  interface AnotherFacet {
    void anotherMethod();
  };
};
```

letados pelo coletor de lixo mesmo ainda estando ativos e aguardando por chamadas. Isso pode ocorrer, por exemplo, caso não haja chamadas por um tempo. Para evitar esse tipo de comportamento e manter os *servants* ativos indefinidamente, é necessário sobrescrever o método "InitializeLifetimeService"que indica quando o objeto deve ser coletado, como descrito no Código 5. Isso deve ser feito em cada *servant* implementado.

Código 5: Implementação da faceta Hello sobrescrevendo o método InitializeLifetimeService

```
  public class HelloImpl : MarshalByRefObject, Hello {
    ...

    public override object InitializeLifetimeService() {
      return null;
    }
  }
```

Para adicionar a faceta implementada no Código 3, no componente, deve-se chamar o método *AddFacet*, que possui como parâmetros o nome, a interface e um objeto que representa a implementação da faceta. O uso desse método pode ser visto no Código 6.

Código 6: Adição de uma Faceta MyFacet a um Componente

```
MarshalByRefObject servant = new MyFacet(context);
string repID = Repository.GetRepositoryID(typeof(MyFacet));

context.AddFacet("MyFacetName", repID, servant);
```

É uma boa prática que toda a faceta possua um construtor que receba o *ComponentContext*. Essa classe pode ser utilizada para acessar outras facetas, o identificados do componente e outros dados que serão descritos na Seção 3.2.

É importante notar que deve-se tomar grande cuidado ao atualizar (*UpdateFacet*) ou remover (*RemoveFacet*) uma faceta. Essas ações devem ser feitas apenas na fase

de construção, antes que o componente esteja sendo utilizado pelos clientes. Alterar as facetas fora da fase de construção pode ser considerada uma mudança na identidade do componente e causar problemas para os clientes que utilizam tais facetas.

3.4 Criação de Receptáculos

Receptáculos representam dependências de interfaces (facetas), e devem ser descritos pelo desenvolvedor da aplicação, não implementados. Eles são manipulados pela faceta básica *IReceptacles*. Se a aplicação desejar manipular seus receptáculos de forma diferente, precisará substituir a implementação da faceta *IReceptacles* através do método *UpdateFacet* do contexto, como descrito na Seção 3.3.

A criação de receptáculos é análoga à criação de facetas, descrita na Seção 3.3. Para adicionar um receptáculo ao componente, deve-se utilizar o método *AddReceptacle*, que espera como parâmetros o nome, a interface esperada e um *boolean* indicando se o receptáculo deve aceitar uma ou múltiplas conexões. O uso desse método pode ser visto no Código 7.

Código 7: Adição de um Receptáculo MyReceptacle a um Componente

```
1 context.AddReceptacle("MyReceptacleName", "IDL:mymodule/MyFacet:1.0", true)
```

3.5 Acesso a Facetas e Receptáculos

A interface *ComponentContext* fornece métodos para acessar tanto as facetas quanto receptáculos. Esses métodos fornecem uma coleção de metadados sobre a faceta ou receptáculo. Exemplos são fornecidos no Código 8.

Código 8: Métodos de Acesso a Facetas e Receptáculos

```
1 Facet facet = context.GetFacetByName("MyFacetName");
2 var facets = context.GetFacets();
3 Receptacle receptacle = context.GetReceptacleByName("MyReceptacleName");
4 var receptacles = context.GetReceptacles();
```

A classe *Facet* que possui os metadados de uma faceta, contém as seguintes informações:

- `Name`: Nome da faceta, fornecido pelo usuário. Atua como o identificador único da faceta dentro do componente.
- `InterfaceName`: A interface IDL da faceta, fornecida pelo usuário.
- `Reference`: O objeto CORBA que representa a faceta.

Um exemplo de como acessar outras facetas de dentro da implementação de uma faceta pode ser visto no Código 9. O exemplo espera que a faceta possua uma referência para o *ComponentContext*, como foi visto no Código 3.

Código 9: Acesso a Outras Facetas de Dentro de Um Método de Faceta

```
1 ...
2 public void myMethod() {
3   Facet anotherFacet = this.context.GetFacetByName("AnotherFacetName");
4   anotherFacet.anotherMethod();
5 }
6 ...
```

A classe *Receptacle* que possui os metadados de um receptáculo, contém as seguintes informações:

- `Name`: Nome do receptáculo. Atua como o identificador único do receptáculo dentro do componente.
- `InterfaceName`: A interface IDL esperada pelo receptáculo.
- `isMultiple`: *Boolean* indicando se o receptáculo aceita múltiplas conexões.
- `GetConnections()`: Lista de conexões desse receptáculo.

4 Builders

Em todos os exemplos anteriores, a definição e "montagem"do componente (adição de facetas e receptáculos) é feita dentro do código fonte. Isso significa que, caso seja necessária alguma mudança nessa configuração, o código-fonte precisa ser alterado. É fácil perceber que essa configuração do componente pode ser definida externamente, permitindo alterações sem a necessidade de mudanças no código-fonte.

Além disso, serviços de mais alto nível podem se beneficiar de descrições em uma linguagem declarativa qualquer, para realizar a implantação automática de componentes num domínio. Administradores de sistema, sem um conhecimento maior sobre o desenvolvimento de componentes de *software*, podem alterar a configuração de aplicações sem a necessidade da intervenção de um programador.

Para facilitar esse processo de externalização da configuração do componente, o SCS utiliza o conceito de *builders*. *Builders* são pequenas bibliotecas que lêem uma descrição de um componente em uma linguagem específica, interpretam os dados para criar um componente de acordo com a configuração desejada. O SCS C# já fornece um *builder* para a linguagem XML.

4.1 XMLComponentBuilder

O *XMLComponentBuilder* interpreta um arquivo XML com a descrição de um componente e retorna um componente pronto para o uso. É possível especificar facetas, receptáculos, o identificador do componente e a implementação do contexto a ser usada. O Código 10 mostra um XML de exemplo, enquanto que o Código 11 demonstra como utilizar o *XMLComponentBuilder*.

O *XMLComponentBuilder* procura, por meio de reflexão, as classes que implementam as facetas. O mecanismo utiliza o valor do elemento *facetImpl* e do atributo *assembly* para identificar a classe descrita no XML.

O SCS fornece em seu pacote de distribuição um arquivo chamado *ComponentDescription.xsd* que contém o *schema* utilizado pelo XMLComponentBuilder.

5 Exemplo Completo

Demonstraremos aqui o uso mais simples de um componente: apenas uma faceta além das três facetas básicas. Não será criado nenhum receptáculo, apesar da existência da faceta *IReceptacles*. Esta demonstração será baseada na demo *Hello*, e exemplos mais complexos poderão ser encontrados nas outras demos do projeto.

O componente *Hello* oferece quatro interfaces: *IComponent*, *IReceptacles*, *IMe-*

Código 10: Arquivo XML Definindo um Componente

```
<?xml version="1.0" encoding="utf-8" ?>
<scs:component xmlns:scs="tecgraf.scs.core"
               xmlns:xi="http://www.w3.org/2001/XMLSchema-instance">
  <id>
    <name>Hello</name>
    <version>1.0.0</version>
    <platformSpec>.Net FrameWork 4.0</platformSpec>
  </id>
  <facets>
    <facet>
      <name>Hello</name>
      <interfaceName>IDL:scs/demos/helloworld/Hello:1.0</interfaceName>
      <facetImpl assembly="HelloServer">Server.HelloServant</facetImpl>
    </facet>
  </facets>
</scs:component>
```

Código 11: Exemplo de uso do XMLComponentBuilder

```
XmlTextReader componentInformation = new XmlTextReader(file);
XMLComponentBuilder builder = new XMLComponentBuilder(componentInformation);
ComponentContext context = builder.build();
```

taInterface e *IHello*, apenas a última faceta é própria do demo. Sua IDL está disponível no Código 12.

Código 12: IDL do Componente Hello

```
module scs{
  module demos{
    module helloworld {
      interface Hello {
        void sayHello();
      };
    };
  };
};
```

O Código 13 utiliza o *XMLComponentBuilder* para realizar a criação do componente. O XML utilizado pode ser visto no Código 10.

Por fim, temos o código "cliente", que acessa o componente. Note que esse código pode ser CORBA puro, não é necessária a criação de um componente para acessar outro componente. Um exemplo de código pode ser visto no Código 14.

Neste exemplo, a mensagem "Hello User!"será exibida somente na máquina servidor. O código cliente apenas terá a chamada *sayHello()* completada corretamente e será finalizado sem erros.

Código 13: Criação do Componente Hello

```
1  static void Main(string[] args) {
2    OrbServices.CreateAndRegisterIiopChannel(0);
3
4    String componentModel = Resources.ComponentDesc;
5    TextReader file = new StringReader(componentModel);
6    XmlTextReader componentInformation = new XmlTextReader(file);
7    XMLComponentBuilder builder = new XMLComponentBuilder(componentInformation);
8    ComponentContext context = builder.build();
9
10   //Escrevendo a IOR do IComponent no arquivo.
11   IComponent component = context.GetIComponent();
12   OrbServices orb = OrbServices.GetSingleton();
13   String ior = orb.object_to_string(component);
14
15   String iorPath = Resources.IorFilename;
16   StreamWriter stream = new StreamWriter(iorPath);
17   try {
18     stream.Write(ior);
19   }
20   finally {
21     stream.Close();
22   }
23 }
```

6 Elementos Adicionais da API do SCS

As seções anteriores descreveram o uso mais comum do SCS para o desenvolvimento de aplicações baseadas em componentes. No entanto, alguns tópicos e funcionalidades adicionais merecem destaque. Nesta seção descreveremos os mais importantes, que podem ser necessários em aplicações ligeiramente mais complexas que o código apresentado anteriormente.

6.1 Extensão do Contexto

Em particular, a classe *DefaultComponentContext* pode ser usada para guardar o estado do componente como um todo, armazenando informações que sejam úteis para mais de uma faceta. Para adicionar informações do estado do componente, o usuário pode optar por estender o *DefaultComponentContext* ou implementar a interface *ComponentContext* e utilizar a classe *DefaultComponentContext* para delegar os métodos que não necessite reimplementar.

Código 14: Utilização do Componente Hello

```
1  static void Main(string[] args) {
2    OrbServices.CreateAndRegisterIiopChannel();
3
4    String helloIorPath = Resources.IorFilename;
5    StreamReader stream = new StreamReader(helloIorPath);
6    String helloIor;
7    try {
8      helloIor = stream.ReadToEnd();
9    }
10   finally {
11     stream.Close();
12   }
13
14   OrbServices orb = OrbServices.GetSingleton();
15   IComponent icomponent = orb.string_to_object(helloIor) as IComponent;
16   Hello hello = icomponent.getFacetByName("Hello") as Hello;
17   hello.sayHello();
18 }
```

6.2 Extensão de Facetas

É possível encontrar necessidade de estender classes que implementam facetas básicas. Por exemplo, a faceta *IComponent*, contém métodos para gerenciar o ciclo de vida do componente, chamados *startup* e *shutdown*. Como a lógica desses métodos deve ficar a cargo do desenvolvedor da aplicação, suas implementações padrão são vazias. Como pode ser visto no Código 15, para que o componente utilize a faceta implementada pelo usuário, é necessário chamar o método *UpdateFacet* visto que o construtor do *DefaultComponentContext* já inicializa as três facetas básicas.

Referências

[1] CORBA - Common Object Request Broker Architecture. `http://www.omg.org/gettingstarted/corbafaq.htm/`.

[2] Iiop.net. `http://iiop.net/`.

[3] MICROSOFT. .net framework. `http://msdn.microsoft.com/en-us/netframework/default.aspx`.

[4] MIDDLEWARE LABORATORY, PUC-RIO. SCS - Software Component System. `http://www.tecgraf.puc-rio.br/scs/`.

Código 15: Extensão da Faceta IComponent

```
1  static void Main(string[] args) {
2  ...
3    string iComponentName = typeof(IComponent).Name;
4    MarshalByRefObject iComponentServant = new MyIComponent(new IComponentServant(context));
5    context.UpdateFacet(iComponentName, iComponentServant);
6  ...
7  }
8
9  public class MyIComponent : MarshalByRefObject, IComponent  {
10     private IComponent iComponent;
11
12     public MyIComponent(IComponent iComponent) {
13       this.iComponent = iComponent;
14     }
15
16     public ComponentId getComponentId() {
17       return iComponent.getComponentId();
18     }
19
20     public MarshalByRefObject getFacet(string facet_interface) {
21       return iComponent.getFacet(facet_interface);
22     }
23
24     public MarshalByRefObject getFacetByName(string facet) {
25       return iComponent.getFacetByName(facet);
26     }
27
28     public void startup() {
29       ....
30     }
31
32     public void shutdown() {
33       ...
34     }
35 }
```