

Tutorial Básico do SDK Java do SCS

Tecgraf

Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)

scs-users@tecgraf.puc-rio.br

1 Introdução

Este documento é um tutorial básico sobre a criação de componentes no modelo SCS v 1.2, utilizando a versão Java da implementação padrão. 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 infraestrutura de execução. Essas informações também podem ser obtidas em outros documentos.

A implementação Java baseada na versão 1.5 da máquina virtual Java em CORBA v 2.3, representada pelo ORB J acorb v 3.1 que está em uso na implementação padrão. Este documento assume que o leitor é familiarizado a conceitos de desenvolvimento de *software* baseado em componentes e a terminologia CORBA.

2 Inicialização do ORB

Para a criação e execução do código de um componente, é necessária a inicialização prévia de um ORB. A instância de ORB criada será passada posteriormente para o construtor de um componente SCS. O procedimento deve ser feito de acordo como código do Código 1.

Código 1: Criação do ORB

```
1
2 public static void main(String[] args) {
3     Properties props = new Properties();
4     orbProps.setProperty("org.omg.CORBA.ORBClass", "org.jacorb.orb.ORB");
5     orbProps.setProperty("org.omg.CORBA.ORBSingletonClass", "org.jacorb.orb.ORBSingleton");
6
7     ORB orb = ORB.init(args, props);
8 }
```

3 Passos Necessários à Criação de um Componente

Aqui 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 representada em Java pela classe `scs.core.ComponentId`. Um identificador de componente contém 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, possívelmente 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".

tação de facet a pode ser conferido no Código 3. Essa facet a precisa ter uma especificação emIDL . Para o nosso exemplo, utilizaremos a IDL contida no Código 4.

Código 3: Implementação de uma Facet a MyFacet

```
1
2 public class MyFacetServant extends MyFacetPOA {
3     public void myMethod() {
4     }
5 }
```

Código 4: Exemplo de IDL de uma Facet a

```
1 module mymodule{
2     interface MyFacet {
3         void myMethod();
4     };
5     interface AnotherFacet {
6         void anotherMethod();
7     };
8 };
```

Essa implementação posteriormente poderá ser instanciada e inserida em um componente como uma nova facet a. Para adicionar uma nova facet a a um componente, o contexto fornece o método *addFacet*, que espera como parâmetros o nome, a interface e a implementação da facet a. O uso desse método pode ser visto no Código 5.

Código 5: Adição de uma Facet a MyFacet a um Componente

```
1
2 ComponentContext context = ...
3 MyFacetServant facetServant = new MyFacetServant();
4 context.addFacet("MyFacetName", MyFacetHelper.id(), facetServant);
```

Por fim, é possível substituir a implementação de uma facet a por uma diferente. Isso é feito através do método *updateFacet*. O método remove a facet a antiga e adiciona a nova, mas mantém o nome e a interface. O Código 6 mostra o uso do método.

É importante notar que deve-se tomar grande cuidado ao atualizar ou remover uma facet a (existe também um método *removeFacet* que não é coberto neste tutorial, mas explicado na documentação da API), pois esses tipos de ação podem levar a resultados inesperados para clientes. Isso pode ser considerado até mesmo como uma mudança em sua própria identidade. Portanto, é recomendada a atualização ou remoção de facet a apenas na fase de construção ou destruição do componente, sempre que suas facet a sejam conhecidas ou estejam sendo utilizadas pelos clientes. Um cuidado ainda maior deve ser tomado em casos onde sejam utilizadas referências persistentes.

Código 6: Atualização de uma Faceta Básica

```
1
2 ComponentContext context = ...
3 MyComponentServant facetServant = new MyComponentServant();
4 context.updateFacet("IComponent", facetServant);
```

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 deseja 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 muito parecida com a de facetas, descrita na Seção ???. Para adicionar um receptáculo a um componente, o contexto fornece o método *addReceptacle*, que espera como parâmetros o nome, a interface esperada e um *boolean* indicando se o receptáculo deve aceitar múltiplas conexões ou somente uma. O uso desse método pode ser visto no Código 7.

Código 7: Adição de um Receptáculo MyReceptacle a um Componente

```
1
2 ComponentContext context = ...
3 MyFacetServant facetServant = new MyFacetServant();
4 context.addReceptacle("MyReceptacleName", MyFacetHelper.id(), true);
```

3.5 Acesso a Facetas e Receptáculos

O contexto fornece métodos para o acesso às suas facetas e receptáculos. O acesso pode ser feito através do nome da faceta ou do receptáculo ou obtendo-se todos de uma vez. Os métodos responsáveis por essas operações são: *getFacetByName*, *getReceptacleByName*, *getFacets*, *getReceptacles*.

4 Exemplo Completo

Demonstremos aqui o uso mais simples de um component e: apenas uma facet a al é m das três facet as básicas. Não será criado nenhum receptáculo, apesar da existência da facet a *Receptacles*. Esta demonstração será baseada na *demo Hello*, e exemplos mais complexos poderão ser encontrados nas outras *demos* do projeto.

O component e *Hello* oferece quatro interfaces: *IComponent*, *IReceptacles*, *IMetaInterface* e apenas uma interface própria, de nome *>Hello*. Sua IDL está disponível no Código 8.

Código 8: IDL do Component e *Hello*

```
1 module scs{
2   module demos{
3     module helloworld {
4       interface Hello {
5         void sayHello();
6       };
7     };
8   };
9 };
```

O Código 9 implementa a facet a *>Hello*, que conta com apenas um método, *sayHello*. Logo em seguida, o Código 10 realiza a criação do component e. O código é bastante similar ao apresentado na Seção 3.3.

Código 9: A Facet a *Hello*

```
1
2 public class HelloServant extends HelloPOA {
3   private String name = "World";
4
5   public void setName(String name) {
6     this.name = name;
7   }
8
9   public void sayHello() {
10    System.out.println("Hello " + name + "!");
11  }
12
13  @Override
14  public org.omg.CORBA.Object _get_component() {
15    return myComponent.getComponent();
16  }
17 }
```

Por fim temos o código "cliente", que acessa o component e. Note que esse código pode ser CORBA puro, não é necessária a criação de um component e para acessar

Código 10: Criação do Componente Hello

```
1
2 public static void main(String[] args) {
3     try {
4         Properties props = new Properties();
5         orbProps.setProperty("org.omg.CORBA.ORBClass", "org.jacorb.orb.ORB");
6         orbProps.setProperty("org.omg.CORBA.ORBSingletonClass", "org.jacorb.orb.ORBSingleton");
7
8         ORB orb = ORB.init(args, props);
9
10        POA poa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
11        poa.the_POAManager().activate();
12
13        ComponentId componentId =
14            new ComponentId("Hello", (byte) 1, (byte) 0, (byte) 0, "java");
15        ComponentContext context = new ComponentContext(orb, poa, componentId);
16
17        HelloServant helloImpl = new HelloServant();
18        helloImpl.setName("User");
19        context.addFacet("Hello", HelloHelper.id(), helloImpl);
20
21        orb.run();
22    }
23    catch (Exception e) {
24        e.printStackTrace();
25        System.exit(1);
26    }
27 }
```

outro componente. Um exemplo desse tipo de código pode ser visto na Listagem 11.

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

5 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.

Código 11: Utilização do Component e Hello

```
1 public static void main(String[] args) {
2     try {
3         Properties props = new Properties();
4         orbProps.setProperty("org.omg.CORBA.ORBClass", "org.jacorb.orb.ORB");
5         orbProps.setProperty("org.omg.CORBA.ORBSingletonClass", "org.jacorb.orb.ORBSingleton");
6
7         ORB orb = ORB.init(args, props);
8
9         String iHelloIOR = ...
10
11         Hello iHelloFacet = HelloHelper.narrow(orb.string_to_object(iHelloIOR));
12         IComponent icFacet = IComponentHelper.narrow(iHelloFacet._get_component());
13
14         icFacet.startup();
15
16         iHelloFacet.sayHello();
17     }
18     catch (Exception e) {
19         e.printStackTrace();
20         System.exit(1);
21     }
22 }
```

5.0.1 Extensão do Contexto

Emparticular, o contexto pode ser usado para guardar o estado do component e como um todo, armazenando informações que sejam úteis para mais de uma facet a. A classe *ComponentContext* já faz isso, guardando todas as informações sobre as facet as e receptáculos. Se o usuário desejar inserir novos dados nessa classe, o ideal é estendê-la.

5.0.2 Extensão de Facetas

Além do exemplo do contexto, é comum também encontrarmos a necessidade de estender classes que implementam facet as. Por exemplo, a classe *scs.core.IComponentServant*, que implementa a facet a *ComponentContext*. *scs.core.IComponentServant* é uma interface (usada (geralmente) para implementar (implementar) a facet a).

5.1 Builders

Em todos os exemplos anteriores, a definição e "montagem" do component e (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 component e 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 utilizar a configuração de aplicações sem a necessidade da intervenção de um programador.

Para facilitar esse processo de externalização da configuração do component e, o SCS utiliza o conceito *builders*. *Builders* são pequenas bibliotecas que têm uma descrição de um component e em uma linguagem específica e então interpretamos dados para criar um component e de acordo com a configuração desejada. O SCS já fornece um *builder* para a linguagem XML.

5.1.1 XMLComponentBuilder

O *XMLComponentBuilder* interpreta um arquivo XML como descrição de um component e e retorna um component e pronto como configuração especificada nesse arquivo. Na versão atual não é possível especificar parâmetros para os construtores das facetas. É possível especificar facetas, receptáculos, o Identificador do Component e e a implementação do contexto a ser usada. O Código 12 mostra um XML de exemplo.

Para obter a implementação de facetas e contexto, *XMLComponentBuilder* carrega as classes ativas do mecanismo de reflexão de Java. O construtor dessas implementações deve oferecer um construtor público que receba um *ComponentContext*. Se o nome de uma faceta já existir, a faceta anterior será substituída pela nova.

O SCS fornece em seu pacote de distribuição um arquivo chamado *Component-Description.xsd* que contém o *schema* XML utilizado pelo XML Component Builder em qualquer linguagem suportada pelo SCS.

Código 12: Arquivo XML Definindo um Component e

```
1 <?xml version="1.0" encoding="iso-8859-1" ?>
2 <component xmlns="tecgraf.scs">
3   <id>
4     <name>ExemploArquivoXML</name>
5     <version>1.0.0</version>
6     <platformSpec>Lua</platformSpec>
7   </id>
8   <context>
9     <type>MyComponentContext</type>
10  </context>
11  <facets>
12    <facet>
13      <name>MyFacetName</name>
14      <interfaceName>IDL:mymodule/MyFacet:1.0</interfaceName>
15      <facetImpl>MyFacet</facetImpl>
16    </facet>
17    <facet>
18      <name>AnotherFacet</name>
19      <interfaceName>IDL:mymodule/AnotherFacet:1.0</interfaceName>
20      <facetImpl>AnotherFacet</facetImpl>
21    </facet>
22  </facets>
23  <receptacles>
24    <receptacle>
25      <name>MyReceptacleName</name>
26      <interfaceName>IDL:mymodule/MyFacet:1.0</interfaceName>
27      <isMultiplex>true</isMultiplex>
28    </receptacle>
29  </receptacles>
30 </component>
```