# SCS Java Tutorial

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

## 1 Introduction

This document is a basic tutorial to the Java version of the SCS component system, focusing on the developer's perspective. It is structured as a high level description of the main parts of the system and intentionally skips some inner details in order to leave the complexity to more specific documentation for advanced users. The system has been implemented with the Java v. 1.5 language and CORBA v. 2.3, which means that the interfaces described in this document use the syntax from the Interface Description Language (IDL) specification from CORBA. This document assumes that the reader is familiar with the concepts related to component software development, the CORBA terminology and the SCS API. A complete tutorial describing the SCS API can be found at `http://www.tecgraf.puc-rio.br/~scorrea/scs/docs.html`

## 2 Abstract Classes

In order to help the application developer to prevent errors introduced by repetitive coding, the component system provides some base classes that implement the common features of the SCS core interfaces. The basic algorithms are implemented in the base classes using template methods, leaving the specific coding for the derived classes. The following topics describe this mechanism.

### 2.1 `IComponentServant`

The `IComponentServant` abstract class implements the most basic features of the `IComponent` interface, providing implementation for the following methods: `getFacet`, `getFacetByName`, `getComponentId`. There are three abstract methods that must be implemented by the user:

- `boolean doStartup()`: this method should implement the specific actions of the startup procedure. The exception declaration and throwing is done by the base class.
- `boolean doShutdown()`: analogous to the previous method, this should define the specific behavior concerning the shutdown procedure.
- `ArrayList<FacetDescription> createFacets()`: this method must return a container (arraylist) of objects describing the facets available. The base class contains the generic algorithm to return the information about facets described in the IComponent interface. This method should initialize the servants that implement the component's facets and fill the associated FacetDescription structure, which is to be inserted in the returned `ArrayList`.

When using the `getFacet` method, one should provide the interface name. This name is a string containing the complete path to the interface, i.e., the interface name and the modules in which it was defined. For example, the interface name of interface defined in listing 1 is `scs::demos::pingpong::PingPong`.

## 2.2 `IMetaInterfaceServant`

`IMetaInterfaceServant` implements the `IMetaInterface` interface, providing implementation for the following methods: `getFacets`, `getFacetsByName`, `getReceptacles` and `getReceptaclesByName`. Analogous to `IComponent-Servant`, when using the `getFacets` method, the complete path to the interface should be provided. Classes deriving from the `IMetaInterfaceServant` must provide two methods to complete the basic features contained in its interface specification:

- `IComponentServant getIComponentServant()`: returns the corresponding `IComponentServant` class that is described by the `IMetaInterface`. The base class contains the code that builds the needed structures to provide the information used by the client application.
- `ArrayList<IReceptaclesServant> getIReceptaclesServants()`: this method is analogous to the previous, but relates to the `IReceptacles` available in the component being described. If the component does not have an `IReceptacles` interface available, this method should return null. The return is an array list of `IReceptaclesServants` because an `IComponent` may have more than one `IReceptaclesServant` available.

### 2.3 `IReceptaclesServant`

`IReceptaclesServant` implements the `IReceptacles` interface, providing implementation for the following methods: `connect`, `disconnect` and `getConnections`. The abstract methods specified by the `IReceptaclesServant` class are:

- `ArrayList<Receptacle> createReceptacles()`: this method instantiates the receptacles provided by the derived class and puts them in an arraylist container. With this container, the base class is able to implement the connection mechanism. The `Receptacle` class is defined in the scs.core.servant package, and its constructor is defined bellow.

  ```
  public Receptacle( String name, String interfaceName,
                     boolean isMultiplex)
  ```

- `int getConnectionLimit()`: this method provides the upper limit on the number of simultaneous connections, to avoid resource exhaustion. This is application specific, so it is delegated to the derived classes.
- `boolean isValidConnection(org.omg.CORBA.Object obj)`: this abstract method should be implemented by the derived class and allows the application to choose the connection validation policy. The boolean return indicates if the connection should be accepted, and in the negative case the InvalidConnection exception is thrown to the client.

# 3 Creating a Component

In this section, a simple component is presented along with the steps to build it. To illustrate the examples, the component chosen is the PingPongDemo, included in the SCS source package.

## Creating the IDL

The PingPong demo is a simple component application that doesn't do anything useful, but demonstrates the basic mechanisms for the SCS system. It consists of a single component with a facet that implements the PingPong interface (listing 1). Two instances of the component are created and connected to each other through their receptacles `infoReceptacle`, as shown in figure 1.

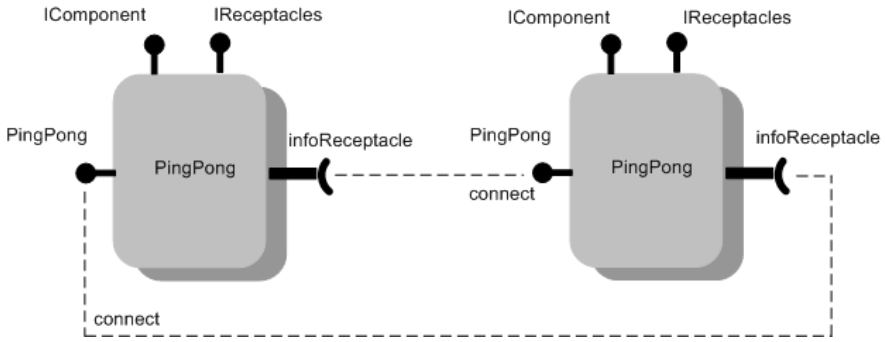


Figura 1: PingPongDemo - two instances of PingPongComponent connected to each other

## Generating the stubs

To create the stubs for the client and server, the following command should be executed, assuming that the Java JDK is on the system's path:

```
idlj -fallTIE pingpong.idl
```

This should generate various files that support the client and server stubs for the application.

Listing 1: The PingPong Interface

```
1  module scs{
2    module demos{
3      module pingpong {
4        interface PingPong {
5          void setId(in long identifier);
6          long getId();
7          void ping();
8          void pong();
9          void start();
10         void stop();
11       };
12     };
13   };
14 };
```

## Deriving abstract methods

In this section we create a component which offers the PingPong interface as a facet. Lets call this component `PingPongComponent`. The first step is to create a new class deriving from `IComponentServant` and provide implementation for the

abstract methods. Listing 2 shows the most important parts of this code. In the listing, lines 10-15 define a facet called `PingPong` which implements the `PingPong` interface. Similarly, lines 18-22 define a facet called `infoReceptacle` which implements the `IReceptacles` interface.

Listing 2: The PingPong Component

```
1  public class PingPongComponent extends IComponentServant {
2
3    private final static String IFACE_PINGPONG = "scs::demos::pingpong::PingPong";
4    private final static String FACET_PINGPONG = "PingPong";
5    private static final String FACET_INFO = "infoReceptacle";
6    private static final String IFACE_INFO = "scs::core::IReceptacles";
7
8    @Override
9    protected ArrayList<FacetDescription> createFacets() {
10     ArrayList<FacetDescription> facets = new ArrayList<FacetDescription>();
11     FacetDescription fd = new FacetDescription();
12     fd.interface_name = IFACE_PINGPONG;
13     fd.name = FACET_PINGPONG;
14     fd.facet_ref = getPingPong(); //returns an object implementing the
15                                   //PingPong interface
16     facets.add(fd);
17
18     fd = new FacetDescription();
19     fd.interface_name = IFACE_INFO;
20     fd.name = FACET_INFO;
21     fd.facet_ref = this.getInfo(); // returns an object implementing the
22             // IReceptacles interface
23     facets.add(fd);
24     return facets;
25   }
26
27   @Override
28   protected boolean doShutdown() {
29     return true;
30   }
31
32   @Override
33   protected boolean doStartup() {
34     return true;
35   }
36
37 }
```

## Creating the servant and the facets

After creating the component, we must provide the servants that implement the component facets. Considering the pingpong application, we must provide a servant for the `PingPong` and `infoReceptacle` facets. Listing 3 shows the implementation of `PingPongServant`, which implements the `PingPong` facet. In line 19, the `infoReceptacle` facet is obtained in order to manage the component connections. In line 22, the object connected to the component receptacle is retrieved in order to call

the `ping` method (line 29). Similarly, the `ping` method obtains the object connected to the component receptacle and calls the `pong` method of the object. Additionally, the `pong` method obtains the object connected to the component receptacle and calls the `ping` method of the object.

## Deploying the Component

The component may be deployed in a .jar file or as a collection of .class files. Either way, it must be registered in the `scs.properties` file, included in directory <SCS-Dir>/scripts/execute. Below, we have a sample from the scs.properties file:

```
component-PingPong-1=scs.demos.pingpong.servant.
                     PingPongComponent
jar-files=pingPong.jar
container.java = ../../scripts/execute/run-container.sh
```

The first line contains the mandatory prefix (component), followed by the name and the version of the deployed component. The right from the equal sign is the full package and class name for the component. The .class files should be in a directory that is listed in the CLASSPATH of the Java environment. The second line should be included only if the component is deployed as a .jar file. The third line is used by the execution node to start a container process. `container.java` is a property that indicates the script to be called when starting a container process. For Windows platform the script to be called is `run-container-cmd`. For Linux platform, the script is `run-container.sh`.

The scripts `run-container.sh` and `run-container.cmd` contains a command line to start the container application. Therefore, before running the examples, set the JAVA_HOME variable in the scripts to reflect the path where java is installed.

## Loading the Component

In this section we describe how containers are created to host component implementations. A client application that instantiates and deploys a `PingPongComponent` component is illustrated in listing 4. First, the application gets a reference to the `ExecutionNode` component representing the node where the component will be executed. In this example, this is done using the JDK Name Service (line 12). Calling method `getFacet`, the application obtains a reference to the `ExecutionNode` facet (line 15), which, in turn, is used to invoke the `startContainer` method (line 25). This method creates and returns a reference to the container component, in which an instance of `PingPongComponent` will be loaded. In the next step, the application gets the container's `ComponentLoader` facet and invokes the `load` method,

passing as parameter the `ComponentId` structure, representing the `PingPongComponent` implementation (lines 30- 39). This method returns a handle to the `PingPongComponent` instance just created. By calling method `startup` (line 40), the instance is activated and calling method getFacet (line 45), the application gets a reference to the `PingPong` facet.

Since the instances were created, the application resolve their dependences, connecting the two instances through their receptacles (lines 53- 66). Finally, the `start` method is invoked in both components in order to start the message exchange.

# 4 Running the example

The PingPong demo can be executed by running the following commands:

1. Run the ORBD. The ORBD is the ORB name service included with the JDK. It is used by the application to locate the Execution Node by name, instead of using the object's IOR. In Windows use the following command to start ORBD:

   ```
   > start "ORBD" orbd.exe  -ORBInitialPort 1050
   -serverPollingTime 200
   ```

   In Linux, use the following command to start ORBD:

   ```
   > orbd -ORBInitialHost localhost -ORBInitialPort 1050
   -serverPollingTime 200
   ```

2. Edit the scs.properties file to set the environment configurations.

3. Edit the run-container scrip (sh or cmd) and set the JAVA_HOME variable to point to the path where java is installed.

4. Run the Execution Node. Considering the SCS-Java root directory (<SCS-DIR>/src/java), the command to start an Execution node is:

   ```
   > java scs.execution_node.servant.ExecutionNodeApp
   ../../scripts/execute/scs.properties
   ```

5. Run the Ping Pong Application. Considering the SCS-Java root directory (<SCS-DIR>/src/java) the command to run the application is:

   ```
   > java scs.demos.pingpong.servant.PingPongApp
   ```

This command creates two component instances in the same machine. In order to run instances in different machines, one should provide the hosts and ports, as described bellow:

```
> java scs.demos.pingpong.servant.PingPongApp
<host-1> <port-1> <host-2> <port-2>
```

# 5 Miscellaneous

The SCS-Java package is provided with some scripts to help running the application. Those scripts are available in the <SCS-DIR>/scripts/execute directory. Also, for Linux platforms, a makefile is provided in directory <SCS-DIR>/src/java. The makefile can be used to compile the SCS-Java package. To use this option, edit the makefile and set the JAVA_HOME variable properly.

# 6 Conclusion

The example shown is just the startup for the developer to get to know with the SCS Java component system. A real example would be much longer and more complex, but the focus would be taken away from the core of the component system. We recommend the execution of the sample code to illustrate the dynamics of the component system and the relationship between its main entities.

Listing 3: Servant implementing the PingPong interface

```
1  public class PingPongServant extends PingPongPOA {
2    private int identifier;
3    private PingPongComponent pingpong = null;
4    private IReceptacles infoReceptacle = null;
5    private ConnectionDescription conns[];
6    private int pingPongCount;
7
8    public  PingPongServant(PingPongComponent pingpong){
9      this.pingpong = pingpong;
10   }
11   public void setId(int identifier){
12     this.identifier = identifier;
13   }
14   public int getId(){
15     return identifier;
16   }
17   public void start() {
18     pingPongCount = 10;
19     infoReceptacle = IReceptaclesHelper.narrow(pingpong.getFacetByName
20         ("infoReceptacle"));
21     try {
22       conns = infoReceptacle.getConnections("PingPong");
23     } catch (InvalidName e) {
24       e.printStackTrace();
25     }
26     if (identifier==1) {
27       for (int i = 0; i < conns.length; i++) {
28         PingPong ppFacet = PingPongHelper.narrow( conns[i].objref );
29         ppFacet.ping();
30       }
31     }
32   }
33   public void stop() {
34     pingPongCount = 0;
35   }
36   public void ping() {
37     for (int i = 0; i < conns.length; i++) {
38       PingPong ppFacet = PingPongHelper.narrow( conns[i].objref );
39       System.out.println("Received ping from " + ppFacet.getId() );
40       ppFacet.pong();
41     }
42   }
43   public void pong() {
44     for (int i = 0; i < conns.length; i++) {
45       PingPong ppFacet = PingPongHelper.narrow( conns[i].objref );
46       System.out.println("Received pong from " + ppFacet.getId() );
47       if( --this.pingPongCount > 0 ) {
48         ppFacet.ping();
49       }
50     }
51   }
52 }
```

Listing 4: PingPong Application

```
public class PingPongApp{
  private static final String EXEC_NODE_NAME = "ExecutionNode";
  private static final String EXEC_NODE_FACET = "scs::execution_node::ExecutionNode";
  private static final String CONTAINER_NAME = "PingPongDemoContainer";
  private ExecutionNode[] execNode = null;
  private IComponent container
  //...

  corbaname = "corbaname::" + host + ":" + port + "#"
    + EXEC_NODE_NAME ;
  try {
    org.omg.CORBA.Object obj = orb.string_to_object(corbaname);
    IComponent execNodeComp = IComponentHelper.narrow(obj);
    execNodeComp.startup();
    Object ob = execNodeComp.getFacet(EXEC_NODE_FACET);
    execNode[i] = ExecutionNodeHelper.narrow(ob);
  } catch (SystemException ex) {
    //...
  }
  try {
    Property prop = new Property();
    prop.name = "language";
    prop.value = "java";
    Property propSeq[] = { prop };
    container = execNode.startContainer(CONTAINER_NAME, propSeq);
    container.startup();
  } catch(ContainerAlreadyExists e){
    // ...
  }
  ComponentLoader loader = ComponentLoaderHelper.narrow(container
    .getFacet("scs::container::ComponentLoader"));
  ComponentId ppCompId = new ComponentId();
  ppCompId.name = "PingPong";
  ppCompId.version = 1;
  ComponentHandle ppHandle = null;

  for (int j=0; j<numComponentPerNode; j++) {
    try {
      ppHandle = loader.load(ppCompId, new String[] { "" });
      ppHandle.cmp.startup();
    } catch (ComponentNotFound e) {
      //...
    }
    PingPong p = PingPongHelper.narrow(ppHandle.cmp
      .getFacetByName("PingPong"));
    p.setId(id);
    if (id ==1)
      pp1Component = ppHandle.cmp;
    else
      pp2Component = ppHandle.cmp;
    id++;
  }
  IReceptacles info1 = IReceptaclesHelper.narrow(pp1Component
    .getFacetByName("infoReceptacle"));
  IReceptacles info2 = IReceptaclesHelper.narrow(pp2Component
    .getFacetByName("infoReceptacle"));
  PingPong pp1 = PingPongHelper.narrow(pp1Component
    .getFacetByName("PingPong"));
  PingPong pp2 = PingPongHelper.narrow(pp2Component
    .getFacetByName("PingPong"));
  try {
    info1.connect("PingPong", pp2);
    info2.connect("PingPong", pp1);
  } catch (InvalidName e) {
    //...
  }
  pp2.start();
  pp1.start();
}
```