Enabling SmartTools components with component technologies: WebServices, CORBA and EJBs.

A summer internship report by
Joseph George Variamparambil

Under the supervision of
Dr. Didier Parigot

At INRIA, Sophia-Antipolis
July, 2002.

Acknowledgement

I would like to express my deepest appreciation and sincere gratitude to Dr. Isabelle Attali and Dr. Didier Parigot for giving me the opportunity to come to INRIA, Sophia-Antipolis and work with the SmartTools team. I would also like to extend special thanks to Alexander Fau and Pascal Degenne for explaining the ins and outs of SmartTools.

Joseph George Variamparambil,
INRIA, Sophia-Antipolis,
July, 2002.
joseph@cse.iitk.ac.in

1. Introduction

With its revised architecture, SmartTools has become even more flexible - The SmartTools components have been redesigned to work independently without relying on the message bus for the exchange of messages between components. The components can now communicate with each other directly and be used as standalone components. The redesign of the architecture led to the study of component technolgies like Webservices, CORBA and EJBs and how these technologies might be used in the SmartTools system to further enhance its flexibility.

An example of a SmartTools component is the Graph component. Its role is to display a graph in a GUI enviornment with methods to add new nodes and edges to the graph. The figure below shows the Graph component in action :

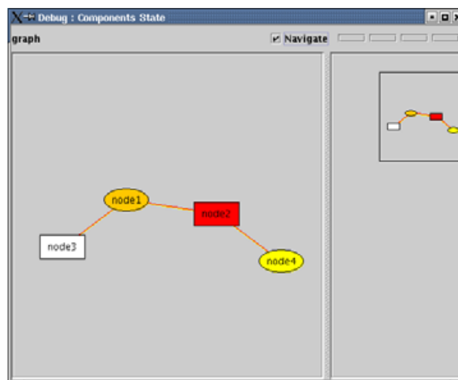

Figure 1: The Graph Component

The main aim of this project was to investigate the various component technologies and to equip the SmartTools components with these technologies. The Graph component was chosen as the experimental component and it was successfully enabled with the component technologies. Detailed instructions to enable any SmartTools component with these technologies is given in the next section.

Each SmartTools component has a 'component description' document. The component description is an XML document that describes the various features of the component and its methods. In order to speed up development, a set of tools was created named CDTools (Component Description Tools). The tools convert the component description to various other descriptions that are required for the different component technologies :

- WSDL for Webservices
- IDL for CORBA
- Home Interface, Remote Interface and the Bean class for EJBs

The CDTools also generate various helper classes that are required to develop and deploy the component services. Instructions on how to use the CDTools are also given in the next section.

After successfully enabling the Graph components with the above technologies, GUI clients were developed to test and demonstrate the component technologies. Listings of the important classes of the webservice, CORBA and EJB clients are shown in the next section.

2. Instructions for enabling component technologies

2.1 Web Services

1. How to install the Tomcat server:

Tomcat is the servlet/JSP container developed by Apache.org.

Steps to install and run Tomcat:-

1. Download the Tomcat v4.0.3 release build binaries (Unix , Windows)
2. Unpack the downloaded file to a directory
3. Set the following enviornment variables:
   - `JAVA_HOME`: the pathname of the directory into which the JDK is installed
   - `CATALINA_HOME`: the pathname of the directory into which Tomcat is installed
4. Starting Tomcat:
   - `%CATALINA_HOME%\bin\startup` (Windows)
   - `$CATALINA_HOME/bin/startup.sh` (Unix)

   After startup, the default web applications included with Tomcat 4.0 will be available by browsing: http://localhost:8080/
5. Stopping Tomcat:
   - `%CATALINA_HOME%\bin\shutdown` (Windows)
   - `$CATALINA_HOME/bin/shutdown.sh` (Unix)

2. How to install Axis:

Apache AXIS is an implementation of the SOAP submission to W3C.

Steps to install Axis:-

1. Install the Tomcat server. (See above)
2. Download the latest release build of axis from Apache's site.
3. Unpack the downloaded file to a directory.
4. Copy the `webapps/axis` directory from the xml-axis distribution into the `webapps` directory of Tomcat (`$CATALINA_HOME/webapps`).
5. Copy `xerces.jar` into `$CATALINA_HOME/webapps/axis/WEB-INF/lib` directory.
6. Start the Tomcat server and browse to http://localhost:8080/axis

3. Creating a SmartTools Component web service

1. Extract the SmartTools component jar to a temporary directory.
   eg: `temp$ jar xvf graph.jar`
2. Include the following into the classpath enviornment variable:
   - . (the current directory)
   - `stComponents/lib/axis.jar`
   - `stComponents/lib/jaxrpc.jar`
   - `stComponents/lib/commons-logging.jar`
   - `stComponents/lib/tt-bytecode.jar`
   - `stComponentslib/wsdl4j.jar`
   - `stComponents/lib/xerces.jar`
   - `stComponents/lib/CDTools.jar`
   - Any jars the component depends on eg: Graph component depends on `koala-graphics.jar` `stComponents/lib/koala-graphics.jar`
3. Change to the resources directory and generate WSDL from the component description as follows:
   eg: `temp/resources$ java CDTools.CD2WSDL graph.xml http://oasis:8080/axis/services/graphSoapPort`

   This will generate the WSDL description of the component. (eg: `graph.wsdl`) in the current directory. The URL indiactes the location of the webservice.

4. Use the `WSDL2Java` tool for building Java proxies and skeletons from WSDL documents:
   eg: `temp/resources$ java org.apache.axis.wsdl.WSDL2Java --output .. --server-side --skeletonDeploy true --deployScope Application graph.wsdl`
   This will generate the java proxies and skeletons in a directory called webService of the SmartTools component.
5. Next, generate the SOAP binding implementation of the component:
   eg: `temp/resources$ java CDTools.CD2SOAPBindingImpl ../graph/webService graph.xml`
   This will generate the SOAP binding implementaion in the directory specified.
6. Compile the generated java sources.
   eg: `temp$ javac graph/webService/*.java`
7. Start the Tomcat server and then deploy the web service using the generated deployment descriptor.
   eg: `temp$ $CATALINA_HOME/bin/startup.sh`
   `temp$ java org.apache.axis.client.AdminClient graph/webService/deploy.wsdd`
8. Create the new webservice-enabled SmartTools component jar.
   eg: `temp$ jar cvf graph.jar *`
9. Copy the new SmartTools component jar into the `webapps/axis/WEB-INF/lib` directory of the Tomcat server.
   eg: `temp$ cp graph.jar $CATALINA_HOME/webapps/axis/WEB-INF/lib/`
10. Copy any jars the component depends on into the `webapps/axis/WEB-INF/lib` directory of the Tomcat server.
    eg: `temp$ cp stComponents/lib/koala-graphics.jar $CATALINA_HOME/webapps/axis/WEB-INF/lib/`
    Now, the SmartTools component webservice is ready to be consumed.
11. **Optional**: To see the SOAP requests and responses use the `tcpmon` utility.
    To use `tcpmon`, change the webservice port in Step #4 to the listen port of `tcpmon` (for example, port 6666).
    So, Step #4 becomes:
    `temp/resources$ java componentDescriptionTools.transform -w graph.xml http://oasis:6666/axis/services/graphSoapPort`
    Carry out the remaing steps (#5-#11) and then run the `tcpmon` utility:
    `temp$ java org.apache.axis.utils.tcpmon 6666 localhost 8080`

4. Creating the client

1. Use the WSDL2Java tool to generate the bindings for the client.
   - `client$ java org.apache.axis.wsdl.WSDL2Java graph.wsdl`
     (Using the WSDL description that was generated from the component xml description.)
   - `client$ java org.apache.axis.wsdl.WSDL2Java http://oasis:8080/axis/services/graphSoapPort?WSDL`
     (if the webservice is running, the WSDL description can be obtained by appending a "?WSDL" to the service URL.)

   This will create a directory (eg: graph/webService) that contains the bindings required for the client.
2. A typical usage of the generated stub classes for the client would be as follows:

```
import graph.webService.*;
  public class Tester {
   public static void main(String [] args) throws Exception {
    // Make a service
    GraphComponentService service = new GraphComponentServiceLocator();
    try{
     // Now use the service to get a stub which implements the SDI.
     GraphPortType port = service.getGraphSoapPort();
     // Make the actual call
     port.newComponentAvailable("Node","oval","red");
    }catch(Exception e){
     e.printStackTrace();
    }
  }
}
```

2.2 CORBA

1. Creating a SmartTools Component CORBA Service

1. Extract the SmartTools component jar to a temporary directory.
   eg: `temp$ jar xvf graph.jar`
2. Include the following into the classpath enviornment variable:
   - . (the current directory)
   - `stComponents/lib/xerces.jar`
   - `stComponents/lib/CDTools.jar`
   - Any jars the component depends on.
     eg: Graph component depends on `koala-graphics.jar`
     `stComponents/lib/koala-graphics.jar`

3. Change to the resources directory and generate IDL from the component description as follows:
   eg: `temp/resources$ java CDTools.CD2IDL graph.xml`
   This will generate the IDL description of the component. (eg: `graph.idl`) in the current directory.
4. Use the `idlj` tool for building stubs and skeletons from IDL:
   `idlj -v -fall -td .. -pkgPrefix corbaService` *`Component_Name IDL_Description`*
   eg: `temp/resources$ idlj -v -fall -td .. -pkgPrefix corbaService graph graph.idl`
   This will generate the java proxies and skeletons in a directory called corbaService of the SmartTools component.
   Note: There should be no method overloading in the component description since idlj does not support method overloading.
5. Next, generate the SmartTools component CORBA server:
   eg: `temp/resources$ java CDTools.CD2CorbaServer .. graph.xml`
   This will generate the SmartTools CORBA server (eg: `graphCorbaServer.java`) in the directory specified.
6. Compile the generated server.
   eg: `temp$ javac graphCorbaServer.java`
7. Start the Java Object Request Broker Daemon, `orbd`, on the server machine:
   `orbd -ORBInitialPort` *`serverPort`* `-ORBInitialHost` *`serverName&`*
   eg: `orbd -ORBInitialPort 1050 -ORBInitialHost oasis&`
8. On the server machine, start the SmartTools component CORBA server as follows:
   eg: `java graphCorbaServer -ORBInitialPort 1050`
   Note: Don't forget to kill the orbd process after clients have finished using it!

2. Creating the CORBA Client

1. Obtain the IDL description of the CORBA service and run `idlj` on it to generate the stubs and skeletons:
   eg: `client$ idlj -v -fall -pkgPrefix corbaService graph graph.idl`
2. Compile the stubs and skeletons:
   eg: `client$ javac graph/corbaservice/*.java`
3. A typical usage of the generated classes for the client would be as follows:

```
import graph.corbaService.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
public class graphClient {
static graphCorba graphImpl;
public static void main(String args[]){
try{
// create and initialize the ORB
ORB orb = ORB.init(args, null);
// get the root naming context
org.omg.CORBA.Object objRef =
orb.resolve_initial_references("NameService");
```

```
// Use NamingContextExt instead of NamingContext. This is
// part of the Interoperable naming Service.
NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
// resolve the Object Reference in Naming
String name = "graph";
graphImpl = graphCorbaHelper.narrow(ncRef.resolve_str(name));
System.out.println("Obtained a handle on server object: " + graphImpl);
graphImpl.newComponentAvailable("aNode","oval","red");
} catch (Exception e) {
System.out.println("ERROR : " + e) ;
e.printStackTrace(System.out);
}
}
}
```

4. To run the client:

```
eg: java graphClient -ORBInitialPort 1050 -ORBInitialHost oasis
```

2.3 EJB

1. Creating the EJB

In order to create the EJB, the J2EE SDK must be installed!

1. Extract the SmartTools component jar to a temporary directory.
   eg: `temp$ jar xvf graph.jar`
2. Include the following into the classpath enviornment variable:
   - . (the current directory)
   - `stComponents/lib/xerces.jar`
   - `stComponents/lib/j2ee.jar`
   - `stComponents/lib/CDTools.jar`
   - Any jars the component depends on.
     eg: Graph component depends on `koala-graphics.jar`
     `stComponents/lib/koala-graphics.jar`
3. Change to the resources directory and generate the ejb interfaces from the component description as follows:
   eg: `temp/resources$ java CDTools.CD2EJB ../ejbService graph.xml`
   This will generate the following files in the specified directory:
   - The Remote Interface: `graphRemoteInterface.java`
   - The HomeInterface: `graphHomeInterface.java`
   - The Enterprise Bean: `graphBean.java`
4. Compile the generated files:
   eg: `ejbService$ javac *.java`

2. Deploying & packaging the EJB

1. In order to deploy & package the EJB use the `deploytool`. Step by step instructions on how to use the tool is described in the "Getting Started" chapter of the J2EE tutorial.
   For example, the tool generates the `graphApp.ear` , the Enterprise Application Jar for the SmartTools graph component with the following application deployment descriptor:
   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <!DOCTYPE application PUBLIC '-//Sun Microsystems, Inc.//DTD J2EE
   Application 1.3//EN' 'http://java.sun.com/dtd/application_1_3.dtd'>
   <application>
   <display-name>graphApp</display-name>
   <description>Application description</description>
   <module>
   <java>app-client-ic.jar</java>
   </module>
   <module>
   <ejb>ejb-jar-ic.jar</ejb>
   </module>
   </application>
   ```
   The ear file contains the ejb-jar and the application-client jar files. The deployment descriptor for the ejb-jar is as follows:
   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise
   JavaBeans 2.0//EN' 'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>
   <ejb-jar>
   <display-name>graphJAR</display-name>
   <enterprise-beans>
   <session>
   <display-name>graphEJB</display-name>
   <ejb-name>graphEJB</ejb-name>
   <home>graphHomeInterface</home>
   <remote>graphRemoteInterface</remote>
   <ejb-class>graphBean</ejb-class>
   <session-type>Stateless</session-type>
   <transaction-type>Bean</transaction-type>
   ```

```
<security-identity>
<description></description>
<use-caller-identity></use-caller-identity>
</security-identity>
</session>
</enterprise-beans>
</ejb-jar>
```

And the deployment descriptor for the application-client:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application-client PUBLIC '-//Sun Microsystems, Inc.//DTD J2EE
Application Client 1.3//EN' 'http://java.sun.com/dtd/application-
client_1_3.dtd'>
<application-client>
<display-name>graphClient</display-name>
<ejb-ref>
<ejb-ref-name>ejb/Graph</ejb-ref-name>
<ejb-ref-type>Session</ejb-ref-type>
<home>graph.ejbService.graphHomeInterface</home>
<remote>graph.ejbService.graphRemoteInterface</remote>
</ejb-ref>
</application-client>
```

Note: The SmartTools component jar and any other jars it depends on should be included in the deployed application's jar by ussing the Add Libary button.

3. Creating the client

1. Detailed instructions on how to create the client are given in the "Creating the J2EE Application Client" chapter of the J2EE Tutorial.
   An example of a standalone client for the graph component is as follows:

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

public class graphClient {

public static void main(String[] args) {
try {
Context initial = new InitialContext();
Context myEnv = (Context)initial.lookup("java:comp/env");
Object objref = myEnv.lookup("ejb/Graph");

graphHomeInterface home =
(graphHomeInterface)PortableRemoteObject.narrow(objref,
graphHomeInterface.class);

graphRemoteInterface Graph = home.create();

Graph.newComponentAvailable("node1","oval","red");

System.exit(0);

} catch (Exception ex) {
System.err.println("Caught an unexpected exception!");
ex.printStackTrace();
}
}
}
```

2. To run the client, use the `runclient` tool as described in the "Running the J2EE Application Client" chapter of the J2EE Tutorial.
   A typical usage of the `runclient` script in a batch file is as follows:

```
set VMARGS=-Dj2eelogin.name=guest -Dj2eelogin.password=guest123
set APPCPATH=graphAppClient.jar
runclient -client graphApp.ear -name graphClient
```

3. The Component Description Tools

The Component Description Tools are a set of five java programs that help in the enabling of SmartTools components for different technologies like Web Services, CORBA and EJBs. The diagram below describes the functionality of the Component Description Tools:-

Installation: Include CDTools.jar and xerces.jar in the classpath.

Web Services

- **CD2WSDL**: This tool generates the equvalent WSDL desccription of the component from the component description.

  Usage: `java CDTools.CD2WSDL` *`Component_Description Service_Location`*
  Where *`Service_Location`* is the URL of the to be SmartTools Component web service.

- **CD2SOAPBindingImpl**: This tool generates the SOAP Binding implementation java source from the component description. The generated file replaces the one generated by the WSDL2Java tool.

  Usage: `java CDTools.CD2SOAPBindingImpl` *`Output_Directory Component_Description`*

CORBA

- **CD2IDL**: This tool generates the equvalent IDL desccription of the component from the component description.

  Usage: `java CDTools.CD2WSDL` *`Component_Description`*
  Note:
  - Only primitive types are supported
  - No method overloading allowed in the component description since idlj does not support it.
- **CD2CorbaServer**: This tool generates the CORBA server java source from the component description.

  Usage: `java CDTools.CD2CorbaServer` *`Output_Directory Component_Description`*

EJB

- **CD2EJB:** This tool generates the interfaces required to build an EJB application. It generates the following files from the component description:
  - Remote Interface
  - Home Interface
  - Bean Class

Usage: `java CDTools.CD2EJB` *`Output_Directory Component_Description`*