[simula . research laboratory]

Work group meeting no. 1
– HelloWorld example

INF5040 (Distributed systems)

Name: Sten L. Amundsen
Date: 31 August 2004

e-mail: stena@simula.no

[simula . research laboratory]

Steps to create a distributed system

1. Create an IDL file with the interface to the CORBA object.
2. Compile the IDL file using the IONA ORBacus IDL-to-Java compiler.
3. Develop servant that implements the interface (server side).
4. Develop the server that starts, stops and administer the servant.
5. Develop a client application.
6. Compile client and server w/servant
7. Start server (that publish it's servant) and client (that uses servant).

simula . research laboratory

Step 1 -Create an IDL file

- Use IDL-to-Java mapping rules to write the file.
- HelloWorld example shows two operations:
  1. request
  2. request with return parameter

```
module helloworld {

    interface Hello {
        void printHello();
        string getHello();

    };//end interface
}; //end package
```

IDL	*Java*
module	package
enum, struct, union	class
interface	interface, class
constant	public static final
exception	class
enum, struct	class

IDL	*Java*
float	float
double	double
long, unsigned long	int
char, wchar	char
boolean	boolean
octet	byte
string, wstring	String

simula . research laboratory

Step 2 - Compile IDL file

- Use the ORBacus IDL-to-Java compiler.
- The jidl compiler generates Java classes and interfaces with the required CORBA-ORBacus hooks implemented.

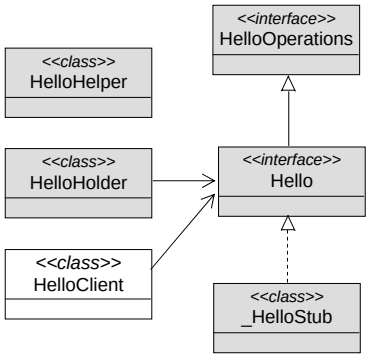
Files	*Description*
Hello	Java interface with same name as the IDL interface identifier (invoked on).
HelloOperations	Defines the operations specified in the IDL-file.
HelloHelper	Utility class for support functions, like narrow object references.
HelloHolder	Contain IDL type references (for out parameters in Java)
HelloPOA	Abstract skeleton class for servant.
_HelloStub	Stub class for the client

C: jidl HelloWorld.idl

simula . research laboratory

Step 2 –Compile IDL –Class diagram -Client

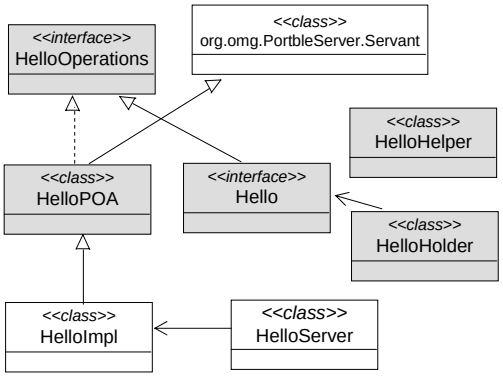
- Relationship between interfaces and classes on client side.
- HelloClient is the application.
- _HelloStub is the interface to the stub.



simula . research laboratory

Step 2 –Compile IDL –Class diagram -Server

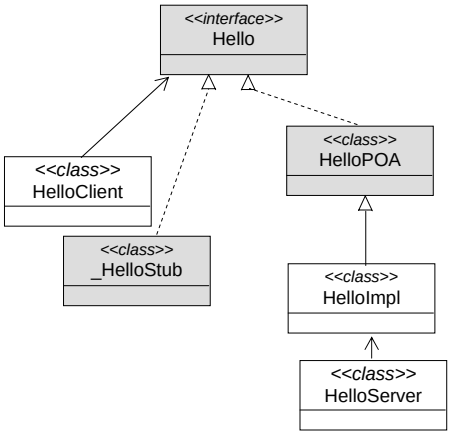
- Relationship between interfaces and classes in server.
- HelloServer is the application that manages the servant.
- _POA is the interface to the skeleton and object adaptor.
- ORB communicates via the *Servant* class (thus, with the –POA)



simula . research laboratory

Step 2 –Compile IDL –Relationship

- When a connection is established between client and servant the relationship is simplified.



simula . research laboratory

Step 3 - Develop servant

- Program a Impl class that inherits _POA (abstract skeleton class).
- Include business logic for the operations stated in the IDL file.

```
package helloworld;

import HelloWorld.HelloPOA;

public class HelloImpl extends HelloPOA {

    HelloImpl() {
    }

    public void printHello() {
        System.out.println(
          "Hello, is there ony one out there.
          ");
    }

    public String getHello() {
        return("Hello.");
    }
}
```

simula . research laboratory

Step 4 - Develop server

- Server has four tasks:
  1. Set the system properties to use ORBacus instead of Java's ORB.
  2. Start ORB and prepare POA.
  3. Instantiate the servant.
  4. Make the servant's IOR (Interoperable Object Reference) general available.

simula . research laboratory

Step 4 - Develop server -Properties

- Use the system properties to the JVM.
- It is possible to set system properties during start-up of the VM, but this often get messy. Also no extra value if CORBA vendor is fixed.

```
java.util.Properties props = System.getProperties();
props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
props.put("org.omg.CORBA.ORBSingletonClass", "com.ooc.CORBA.ORBSingleton");
```

simula . research laboratory

Step 4 - Develop server –Start ORB and POA

- The ORB is started when the class method init() is invoked. Since a multi thread server, it will now handle itself.
- Input parameters is the system properties (after replacing Java's CORBA with ORBacus) and arguments (none in this case).
- Use RootPOA and it's manager object, i.e. no need to introduce our own POA.

```
try {
   //ORB
   String[] arguments = null;
   org.omg.CORBA.ORB orb = ORB.init(arguments, props));
   //POA
   org.omg.CORBA.Object poaObj = orb.resolve_initial_references("RootPOA");
   org.omg.PortableServer.POA rootPOA = POAHelper.narrow(poaObj);
   org.omg.PortableServer.POAManager manager = rootPOA.the_POAManager();
   manager.activate();
} catch (Exception ex) {}
```

simula . research laboratory

Step 4 - Develop server –Instantiate the servant

- Step 3 developed the servant. Now the ORB and POA is ready, so the servant can be instantiated and made available to the POA and ORB.

```
HelloImpl helloImplementation = new HelloImpl();
Hello hello = helloImplementation._this(orb);
// Object reference can be stored in file, e-mail, directory server
// or preferably a CORBA naming service.
```

simula . research laboratory

Step 4 - Develop server –Store IOR

- The object reference can be stored in numerous ways.
- HelloWorld example uses a file “Hello.ref”.

```
try {
     String reference = orb.object_to_string(hello);
     String refFile = "Hello.ref";
     PrintWriter out = new PrintWriter(new FileOutputStream(refFile));
     out.println(reference);
     out.close();
 } catch (IOException ioex) {}
```

simula . research laboratory

Step 5 - Develop a client application

- The client has five tasks:
  1. Set the system properties to use ORBacus instead of Java’s ORB.
  2. Start the ORB.
  3. Instantiate the client object.
  4. Get reference to it’s servant on the server.
  5. Invoke business operations on servant.

1-3 similar to the servant side, thus, not repeated.

simula . research laboratory

Step 5 - Develop client application –Get reference

- The object reference shall be retrieved from the file "Hello.ref".
- Locate file and read the text string.

```
try {
   String refFile = "Hello.ref";
   java.io.BufferedReader in =
        new BufferedReader(new FileReader(refFile));
   String ref = in.readLine();
   in.close();

   org.omg.CORBA.Object corbaObj = orb.string_to_object(ref);

} catch (IOException ioex) {}
```

simula . research laboratory

Step 5 - Develop client application –Invoke business operations

- Ordinary casting doesn't work. Instead use Helper class to narrow object reference to the interface.
- Call operations on interface……..

```
// Since corbaObj is to remote machine, the Helper class returns a
// a reference to the stub (_HelloStub).
Hello hello = HelloHelper.narrow(corbaObj);

hello.printHello();
String resultHello = hello.getHello();
System.out.println("Received from server: " +resultHello);
```

simula . research laboratory

Step 6 – Compile client and server

- Depend on development environment.
- Use **src** and **class** catalogues to keep things tidy.

UNIX users at IFI,
only LINUX machines.
JDK OK so is CORBA.

```
javac -sourcepath src/helloworld -d class src/helloworld/*.java
```

simula . research laboratory

Step 7 – Start client and server

- Remember correct environment…..

UNIX users at IFI,
only LINUX machines.
JDK OK so is CORBA.

```
>java -version

>java helloworld.HelloClient

>java helloworld.HelloServer
```

NT, Windows, XP can
use a bat file to ensure
stable environment.

```
set JAVA_HOME=C:\Java\SDK\v1.3.1_08
set CORBA_HOME=C:\software\IONA\JOB_4_1_2
set CLASSPATH=.;%CORBA_HOME%\lib\OB.jar
set JAVA=%JAVA_HOME%\bin\java -classpath %CLASSPATH%

%JAVA% helloworld.HelloClient

%JAVA% helloworld.HelloServer
```

simula . research laboratory

HelloWorld example

- To understand the implementation, refer to the helloworld example on the course home page.
- All students shall develop their own implementation using the provided IDL.

IDL file:

- /src/helloworld.idl

Java source files:

- /src/helloworld

Javadoc:

- /javadoc/helloworld/index.html