Coppelia: Observer / Observable Interface

Last updated: August 7, 2000

Javelin Copyright Statement

Copyright

1.0 About this Document

One of the most powerful interfaces that is available to the Coppelia user is the Java Observer/Observable interface, or otherwise know as the "In Process" interface.

This interface allows a user to run a Coppelia FIX engine and a client program both in the same Java Virtual Machine space - allows them to run "in process".

The advantage to this is that it is FAST. Some of the best performance numbers that we have seen have been with the Observer/Observable interface.

This document describes the procedures necessary to use this feature.

2.0 .dat File Entry

To use this interface, you need to make a change to the INTERFACE setting in the .dat file.

2.1 For Coppelia 4.1e or 4.2a

```
INTERFACE          OBSERVER
```

2.2 For Coppelia 4.1f or 4.2b

```
interface_type     INPROC
```

3.0 Modification of Start Up Script

Using the Observer/Observable interface requires a different method of starting the Coppelia engine. For most every other interface, the **go_buy**/**go_sell** files call the 'java' command with a parameter of Coppelia. For example -

```
java -classpath %CLASSPATH% Coppelia buy.dat
```

to start up the Coppelia engine using the CORBA interface.

However, in Observer/Observable, all Coppelia functionality will be contained in the user-created client application code. Starting the Coppelia engine will require compilation and execution of the client application code. For example, if the user had created a client program called **ObserverExample.java**, a sample NT **go_buy** script would look somewhat like this:

```
SET PATH=D:\jdk1.1.8\bin;
SET CLASSPATH=D:\testing\coppelia\classes\pro.zip;
D:\testing\coppelia\classes\coppelia.jar;
D:\testing\coppelia\classes\mct3_0.zip;
D:\testing\coppelia\classes\rogue.zip;
D:\testing\coppelia\classes\OrbixWeb31c.jar;
D:\testing\coppelia\lib;

javac ObserverExample.java

java ObserverExample buy.dat
```

The **javac** line compiles the java program, and the **java** line executes the program. The ObserverExample program takes one argument, which is the **buy.dat** file (or whichever .dat file the user wants the program to use).

Note: The **classpath** setting must contain every file that Coppelia needs to run - including files like **pro.zip**, **OrbixWeb31c.jar** (even though the CORBA interface is not being used), etc.

4.0 Methods to Use

There are a number of methods that a client application will would use in order to work with Coppelia's Observer / Observable interface:

Application Related

1. **CoppeliaSrvFactory.getInProcObject()** to start the Coppelia process or obtain a handle to it subsequently. An example:

   ```
   CoppeliaSrv srv = CoppeliaSrvFactory.getInProcObject(args);
   ```

   where **args** contains the name of the .dat file that the Observer/Observable program is going to be run with.

2. **addListener()** of CoppeliaSrv to register a callback for application messages and **addMonitor()** for session connectivity and statistical information and notification. An example:

   ```
   MyObserver l = new MyObserver();
   try {
       srv.addListener(l);
   } catch(Exception e) {}
   ```

   where **l** is the MyObserver object (MyObserver is another class that is defined elsewhere in the java code). See the **ObserverSession.java** program in Appendix C to see what the MyObserver class looks like.

3. **post()** of CoppeliaSrv for outgoing messages. An example:

   ```
   res = srv.post(o);
   ```

   where **'o'** is some kind of message object that has been created prior to this function call. See Appendix B - **ObserverOrders.java** for an example the usage of post.

Operations Related

1. **`operatorCommand()`** of CoppeliaSrv. An example:

   ```
   srv.operatorCommand("connect all");
   ```

   which calls the 'connect all' command to connect all remote counter parties.

2. **`getOperatorData()`** of CoppeliaSrv. An example:

   ```
   OperatorData OpData[] = srv.getOperatorData();
   ```

   which retrieves a complete array of operator data about the coppelia engine (all the information retrieved is that same as see in the Coppelia FIXometer product, or in the **`CSRemoteData.idl`** file).

Because the client application 'lives' in the same process space as the Coppelia process, the Coppelia process is actually started by the client application. This is usually the first thing to do in the client application. This is done through the use of the factory method:

`getObservableObject()`

An example:

```
CoppeliaSrv srv = CoppeliaSrvFactory.getInProcObject(args);
```

The **`args`** here is actually the configuration file, for example **`buy.dat`**. The factory method will only create one instance of Coppelia, and therefore it is Singleton. Your client application could invoke this method the second time by providing an empty **`String[]`** to obtain the handle to **`CoppeliaSrv`**.

Because we are talking Observer/Observable, you need to develop a listener that will handle the incoming messages, and register that listener. For example,

```
AppObserver l = new AppObserver();
try {
    srv.addListener(l);
} catch(Exception e) {}
```

Once the listener is registered, and when there is an incoming FIX message, the listener will be notified with an object as the argument to **`update(Observable o, Object arg)`** in the implementation of the Observer. The object is of MessageObject type which is the FIX Application object. For FIX application objects, please refer the corresponding IDL files. Note that the **`get()`** method of **`CoppeliaSrv`** interface for Observer/Observable interface is not currently supported.

5.0 Session Connectivity

There is an **`addMonitor()`** method that allows a client application to register a listener for monitoring details about a FIX session. This listener will be notified when a session is down. The object passed in to this listener is of type **`OperatorData`**. For detail of the data structure of OperatorData, please refer to **`CSRemoteData.idl`** because it has the identical structure.

In order to use this feature, please add a flag in the .dat file:

```
SESSION_NOTIFICATION          ON
```

The **`connect_state`** in **`OperatorData`** is one of the following:

```
public static final int DISCONNECTED = 0;
public static final int SESSION_CONNECTION = 1;
public static final int APPLICATION_CONNECTION = 2;
public static final int LOGGING_ON_CONNECTION = 3;
```

Please refer to the examples for usage.

6.0 Outgoing Messages

For outgoing messages, the **`post()`** method of **`CoppeliaSrv`** is used. Since this is in the same process of Coppelia, and because Java uses references, you need to create a new application object every time you post, which is the common practice, that is, no one Order is the same. This is only note for your information.

7.0 Operator's API

operatorCommand()

This method allows a client application to perform administrative operations such as connect, disconnect, exit, eod, etc. This works basically the same way as if you would type these commands at the command prompt of Coppelia.

getOperatorData()

This operator method allows a client application to query the statistics of a certain connection. The method returns an array of **`OperatorData`**. The structure of **`OperatorData`** is the same as described is **`CSRemoteData.idl`**. Please refer to this file for more details.

For detail of usage, please refer to the ObserverExample.

Appendix A: Basic Observer/Observable example - SimpleObserver.java

The example shown here provides an extremely basic method of using the Observer/ Observable interface. This simple program starts up the Coppelia FIX engine and then runs the 'stats' command, using the operatorCommand method.

```
import com.javtech.coppelia.*;
import com.javtech.coppelia.interfaces.*;
import java.util.*;

public class SimpleObserver
{
    public static void main(String[] args)
    {
      System.out.println("Usage: java  SimpleObserver <buy.dat>");

      /* Create the CoppeliaSrv object based upon the dat file
         that is read in */

      CoppeliaSrv srv = CoppeliaSrvFactory.getInProcObject(args);
      System.out.println("Started Server");

  try { Thread.sleep(10000); } catch(Exception e) {
  e.printStackTrace(); }

  /* Call the Coppelia stats command to show the status of all the
      connections */

      try {
                  String stats1 = srv.operatorCommand("stats");

                  System.out.println(" *** STATS *** ");
                  System.out.println("" + stats1);

            } catch(Exception e) {}

    }
}
```

The following is a batch file that will compile and run this program on Windows NT.

```
SET PATH=D:\jdk1.1.8\bin;
SET CLASSPATH=D:\testing\coppelia\classes\pro.zip;
D:\testing\coppelia\classes\coppelia.jar;
D:\testing\coppelia\classes\mct3_0.zip;
D:\testing\coppelia\classes\rogue.zip;
D:\testing\coppelia\classes\OrbixWeb31c.jar; D:\jdk1.1.8\lib;

javac SimpleObserver.java

java SimpleObserver buy.dat
```

Make sure to have the actual Coppelia engine's .dat file in the same directory as the **`SimpleObserver.java`** program.

Appendix B: Sending Messages via Observer/Observable - ObserverOrders.java

This example simply sends 10 FIX order messages to a counter party. It creates the order object using this command:

```
Order o = new Order();
```

It then populates the required fields within that order, then sends it using the **post()** command, like shown below:

```
res = srv.post(o);
```

The batch file to run this program would be no different than the one you used to run **SimpleObserver.java** in Appendix A.

```
import com.javtech.coppelia.*;
import com.javtech.coppelia.interfaces.*;
import java.util.*;

public class ObserverOrders
{
      public static void main(String[] args)
      {
            System.out.println("Usage: java  ObserverOrder <buy.dat>");

            CoppeliaSrv srv = CoppeliaSrvFactory.getInProcObject(args);

            try {
                  Thread.sleep(10000);
            }
            catch(Exception e) {
                  e.printStackTrace();
            }

            for(int i=0; i<10; ++i) {

                  int res = 0;

                  //create the Order object. See COrder.idl file for
                  //tag names and data types

                  Order o = new Order();

                  o.header.TargetCompID = "SBI";
                  o.header.SenderSubID = "JOE";

                  o.Symbol = "ADVS";    //** ticker
                  o.Side = "1";         //** 1=BUY, 2=SELL
                  o.OrderQty = 1000;
                  o.HandlInst = "1";    //** 1=best execution
```

```
                o.OrdType = "1";      //** 1=MKT, 2=LMT

                o.ClOrdID = "" + (i+1);   //** this must be a unique
identifier
                try {

                res = srv.post(o);
            System.out.println("Result of posting order : " + o.ClOrdID
+ " is : " + res);

                }catch(Exception e) {e.printStackTrace();}
            }

      }
}
```

Appendix C: Session Notification - ObserverSession.java

This example provides notification to the user when a remote connection's state changes - either when it connects or when it disconnects.

Remember to set

SESSION_NOTIFICATION ON

in the .dat file for this to work properly!

```
import com.javtech.coppelia.*;
import com.javtech.coppelia.interfaces.*;
import java.util.*;

// This is a sample Observer implementation.
class MyObserver implements Observer
{

      public MyObserver() { super(); }

    public void update(Observable o, Object obj) {
            if (obj instanceof OperatorData)
            handleOperatorData((OperatorData)obj);

  }

  public void handleOperatorData(OperatorData odata) {
    // Print the type of connection change and the target's ID
    if (odata.connect_state != ConnectState.DISCONNECTED)
      System.out.println(" *** Target " + odata.firm_id + "
connected.");
    else
      System.out.println(" *** Target " + odata.firm_id + "
disconnected.");

  }
}

// Sample application.
public class ObserverSession
{
    public static void main(String[] args)
    {
      System.out.println("Usage: java  ObserverExample <buy.dat>");


      CoppeliaSrv srv = CoppeliaSrvFactory.getInProcObject(args);
      System.out.println("Started Server");

  try { Thread.sleep(10000); } catch(Exception e) {
e.printStackTrace(); }
```

```
        System.out.println("Added listener");

        // Add the listener, i.e., register the Observer.
        MyObserver l = new MyObserver();
        try {
            srv.addListener(l);
        } catch(Exception e) {}

        try {
            srv. addMonitor(l);
        } catch(Exception e) {}

    }
}
```

Appendix D: Using the "Operator Command" function - ObserverOpCommand.java

This example shows how to use the Observer Observable operator commands.

One thing to note with the Operator commands is that they can be called by themselves, or they can be called to return something.

for example, a user could call

```
srv.operatorCommand("connect all");
```

but this command does not return anything. It would be up to the user to check the status of the session to see if it is up or not.

In the case that a function actually does have some output, for example, 'stats', a user could use:

```
String stats_output = srv.operatorCommand("stats");
System.out.println("Stats is : " + stats_output);
```

to actually print out the return of the stats command.

Any Coppelia command line option is available to this command, for example:

`connect, disconnect, eod, stats, msg_seq_num_in, msg_seq_num_out,` etc.

The following example does the following:

- Connects a remote counter party called SBI
- Calls stats
- Disconnects that counter party
- Runs end of day
- Calls stats again

```
import com.javtech.coppelia.*;
import com.javtech.coppelia.interfaces.*;
import java.util.*;

public class ObserverOpCommand
{
      public static void main(String[] args)
      {
            System.out.println("Usage: java  ObserverOrder <buy.dat>");

            CoppeliaSrv srv = CoppeliaSrvFactory.getInProcObject(args);

            try {Thread.sleep(5000); }
            catch(Exception e) {e.printStackTrace();}

            System.out.println("Connecting remote counter party SBI!");

            try {
                  srv.operatorCommand("connect SBI");
            } catch(Exception e){  }

            try {Thread.sleep(5000); }
            catch(Exception e) {e.printStackTrace();}

            System.out.println("Calling stats");

            try {
            String stats_output = srv.operatorCommand("stats");
            System.out.println(" *** Stats output *** : " +
stats_output);
            } catch(Exception e){  }



            try {Thread.sleep(5000); }
            catch(Exception e) {e.printStackTrace();}

            System.out.println("disconnecting remote counter party
SBI!");

            try {
            srv.operatorCommand("disconnect SBI");
            } catch(Exception e){  }

            try {Thread.sleep(5000); }
            catch(Exception e) {e.printStackTrace();}

            System.out.println("Running end of day on counter party
SBI!");

            try {
            srv.operatorCommand("eod SBI");
            } catch(Exception e){  }

            try {Thread.sleep(5000); }
            catch(Exception e) {e.printStackTrace();}
```

```
            System.out.println("Calling stats");

            try {
            String stats_output = srv.operatorCommand("stats");
            System.out.println(" *** Stats output *** : " +
stats_output);
            } catch(Exception e){  }

            try {Thread.sleep(5000); }
            catch(Exception e) {e.printStackTrace();}

      }
}
```

Appendix E: Receiving Messages - ObserverReceive.java

This example shows how to receive and process messages in Observer/Observable.

When receiving messages, a user will always receive first a generic object that could contain any kind of FIX message (Order, IOI, Execution Report, etc). A user then must cast that generic object into the specific FIX message object. In this example, if a message comes in, it is in generic 'Object' format:

```
public void update(Observable o, Object arg)
```

A user must then determine what kind of object it is. One way to do this is by looking at the MsgType field in the header. However - the Object needs to be cast to a generic **`MessageObject`** class (The MessageObject class is a superset of all the FIX messages) . NOTE: since every FIX message contains a header and a trailer, the MessageObject class allows you to access that information! But you cannot access any of the information in the body of the message while the Message is in MessageObject format - i.e. you cannot access the Symbol in an Order message while it is still in MessageObject format.

For example, to cast an Object arg to a MessageObject, do:

```
MessageObject message = (MessageObject)arg
```

To look at the MsgType in that MessageObject, use:

```
System.out.println("The message type of the incoming message is : " +
message.header.MsgType);
```

Once you know what kind of message it is, you can then cast the MessageObject to that specific message type, and process it accordingly:

```
if (message.header.MsgType.equals("D")) {
      Order ord1 = (Order)message;
      …
      }
```

As with the other examples, this program only takes one argument (the name of the batch file).

```
import com.javtech.coppelia.*;
import com.javtech.coppelia.interfaces.*;
import java.util.*;

class MyObserver implements Observer
{

    public MyObserver() { super(); }

    public void update(Observable o, Object arg)
```

```
    {

      System.out.println("Message came in:");

      MessageObject message = (MessageObject)arg;

      System.out.println("The message type of the incoming message is :
" + message.header.MsgType);

      if (message.header.MsgType.equals("D")) {
                           Order ord1 = (Order)message;
            System.out.println("Order Symbol is : " + ord1.Symbol);
                        }

      else if (message.header.MsgType.equals("8")) {
            ExecutionReport exec1 = (ExecutionReport)message;
      System.out.println("Execution Report Symbol is : " +
exec1.Symbol);
                        }

      else if (message.header.MsgType.equals("6")) {
                           IndicationOfInterest ioi1 =
(IndicationOfInterest)message;
                  System.out.println("IOI ID is : " + ioi1.IOIID);
                        }

      else {System.out.println("Unhandled message type : " +
message.header.MsgType); }

    }
}

public class ObserverReceive
{
    public static void main(String[] args)
    {
      System.out.println("Usage: java  ObserverReceive <buy.dat>");

      CoppeliaSrv srv = CoppeliaSrvFactory.getInProcObject(args);
      System.out.println("Started Server");

  try { Thread.sleep(10000); } catch(Exception e) {
e.printStackTrace(); }
      System.out.println("Added listener");

      // Add the listener, i.e., register the Observer.
      MyObserver l = new MyObserver();
      try {
          srv.addListener(l);
      } catch(Exception e) {}

      }

}
```

Appendix F: Getting Operator - ObserverOpData.java

This example shows how to retrieve Operator Data (connection status, message sequence number in/out, etc.)

Note that you can look at the **`CSRemoteData.idl`** file to find out all the information that can be retrieved from the Operator Data object.

This example retrieves the connection status as well as the message sequence number in and out.

Note that the target Company ID that is looked at is specified directly in the code:

```
String target = "SBI";
```

This can be specified to be any Company ID, but it is a remote ID that the Coppelia engine connects to - it is NOT the local Firm ID.

```
import com.javtech.coppelia.*;
import com.javtech.coppelia.interfaces.*;
import java.util.*;

public class ObserverOpData
{
      public static void main(String[] args)
      {
            System.out.println("Usage: java  ObserverOpData
<buy.dat>");

            CoppeliaSrv srv = CoppeliaSrvFactory.getInProcObject(args);

            try {Thread.sleep(10000); }
            catch(Exception e) {e.printStackTrace();}


      try {
                        System.out.println("Checking operator data!");

                        OperatorData OpData[] = srv.getOperatorData();

                        /* Find the specific target ID in the list */
                        String target = "SBI";
                        System.out.println("Finding Operator Data for
target : " + target);


      for (int x=0; x<OpData.length; x++) {


            if (OpData[x].firm_id.equals(target)) {
            System.out.println("Found target!");
```

```
            System.out.println("Displaying Operator Data!");

            System.out.println("Status of connection to : " + target +
" is : " + OpData[x].connect_state);
            System.out.println("Seq num in for : " + target + " is : "
+ OpData[x].msg_sequence_num_in);
            System.out.println("Seq num out for : " + target + " is : "
+ OpData[x].msg_sequence_num_out);
            }
                        }
                  }
                   catch(Exception e) {
      System.out.println("Problems with operator data" +
e.getMessage());
                    }

      }
}
```