Object Composition

Interfaces support single and multiple inheritance, and they are all based on com.sun.star.uno.XInterface. In the API reference, this is mirrored in the *Base Hierarchy* section of any interface specification. If you look up an interface, always check the base hierarchy section to understand the full range of supported methods. For instance, if you look up com.sun.star.text.XText, you see two methods, `insertTextContent()` and `removeTextContent()`, but there are nine more methods provided by the inherited interfaces. The same applies to exceptions and sometimes also to structs, which support single inheritance as well.

The service specifications in the API reference can contain a section *Included Services* , which is similar to the above in that a single included old-style service might encompass a whole world of services. However, the fact that a service is included has nothing to do with class inheritance. In which manner a service implementation technically includes other services, by inheriting from base implementations, by aggregation, some other kind of delegation or simply by re-implementing everything is by no means defined (which it is not, either, for UNO interface inheritance). And it is uninteresting for an API user – he can absolutely rely on the availability of the described functionality, but he must never rely on inner details of the implementation, which classes provide the functionality, where they inherit from and what they delegate to other classes.

UNO Concepts

Now that you have an advanced understanding of OpenOffice.org API concepts and you understand the specification of UNO objects, we are ready to explore UNO, i.e. to see how UNO objects connect and communicate with each other.

UNO Interprocess Connections

UNO objects in different environments connect via the interprocess bridge. You can execute calls on UNO object instances, that are located in a different process. This is done by converting the method name and the arguments into a byte stream representation, and sending this package to the remote process, for example, through a socket connection. Most of the examples in this manual use the interprocess bridge to communicate with the OpenOffice.org.

This section deals with the creation of UNO interprocess connections using the UNO API.

Starting OpenOffice.org in Listening Mode

Most examples in this developers guide connect to a running OpenOffice.org and perform API calls, which are then executed in OpenOffice.org. By default, the office does not listen on a resource for security reasons. This makes it necessary to make OpenOffice.org listen on an interprocess connection resource, for example, a socket. Currently this can be done in two ways:

- Start the office with an additional parameter:

```
soffice -accept=socket,host=0,port=2002;urp;
```

This string has to be quoted on unix shells, because the semicolon '`;`' is interpreted by the shells

- Place the same string without `'-accept='` into a configuration file. You can edit the file

  <OfficePath>/share/registry/data/org/openoffice/Setup.xcu

  and replace the tag

```
<prop oor:name="ooSetupConnectionURL"/>
```

with

```
<prop oor:name="ooSetupConnectionURL">
  <value>socket,host=localhost,port=2002;urp;StarOffice.ServiceManager
  </value>
  </prop>
```

If the tag is not present, add it within the tag

```
<node oor:name="Office"/>
```

This change affects the whole installation. If you want to configure it for a certain user in a network installation, add the same tag within the node `<node oor:name="Office/>` to the file *Setup.xcu* in the user dependent configuration directory *<OfficePath>/user/registry/data/org/openoffice/*

Choose the procedure that suits your requirements and launch OpenOffice.org in listening mode now. Check if it is listening by calling *netstat -a* or *-na* on the command-line. An output similar to the following shows that the office is listening:

```
TCP <Hostname>:8100 <Fully qualified hostname>: 0 Listening
```

If you use the *-n* option, *netstat* displays addresses and port numbers in numerical form. This is sometimes useful on UNIX systems where it is possible to assign logical names to ports.

If the office is not listening, it probably was not started with the proper connection URL parameter. Check the Setup.xcu file or your command-line for typing errors and try again.

Note – Note: In versions before OpenOffice.org 1.1, there are several differences. The configuration setting that makes the office listen every time is located elsewhere. Open the file *<OfficePath>/share/config/registry/instance/org/openoffice/Setup.xml* in an editor, and look for the element:

```
<ooSetupConnectionURL cfg:type="string"/>
```

Extend it with the following code:

```
<ooSetupConnectionURL cfg:type="string">
 socket,port=2083;urp;
 </ooSetupConnectionURL>
```

The commandline option -accept is ignored when there is a running instance of the office, including the quick starter and the online help. If you use it, make sure that no soffice process runs on your system.

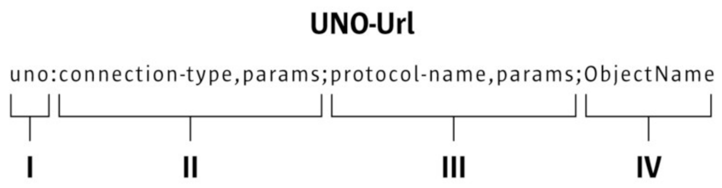
The various parts of the connection URL will be discussed in the next section.

Importing a UNO Object

The most common use case of interprocess connections is to import a reference to a UNO object from an exporting server. For instance, most of the Java examples described in this manual retrieve a reference to the OpenOffice.org `ComponentContext`. The correct way to do this is using the com.sun.star.bridge.UnoUrlResolver service. Its main interface com.sun.star.bridge.XUnoUrlResolver is defined in the following way:

```
interface XUnoUrlResolver: com::sun::star::uno::XInterface
  {
      /** resolves an object on the UNO URL */
      com::sun::star::uno::XInterface resolve( [in] string sUnoUrl )
          raises (com::sun::star::connection::NoConnectException,
                  com::sun::star::connection::ConnectionSetupException,
                  com::sun::star::lang::IllegalArgumentException);
  };
```

The string passed to the `resolve()` method is called a *UNO URL*. It must have the following format:



A UNO URL scheme

An example URL could be *uno:socket,host=localhost,port=2002;urp;StarOffice.ServiceManager*. The parts of this URL are:

- The *URL schema uno:*. This identifies the URL as UNO URL and distinguishes it from others, such as *http:* or *ftp:* URLs.
- A string which characterizes the *type of connection* to be used to access the other process. Optionally, directly after this string, a comma separated list of name-value pairs can follow, where name and value are separated by a '='. The currently supported connection types are described in Opening a Connection. The connection type specifies the transport mechanism used to transfer a byte stream, for example, TCP/IP sockets or named pipes.
- A string which characterizes the *type of protocol* used to communicate over the established byte stream connection. The string can be followed by a comma separated list of name-value pairs, which can be used to customize the protocol to specific needs. The suggested protocol is urp (UNO Remote Protocol). Some useful parameters are explained below. Refer to the document named UNO-URL at udk.openoffice.org for the complete specification.
- A process must explicitly export a certain object by a distinct name. It is not possible to access an arbitrary UNO object (which would be possible with IOR in CORBA, for instance).

The following example demonstrates how to import an object using the UnoUrlResolver:

```
XComponentContext xLocalContext =
      com.sun.star.comp.helper.Bootstrap.createInitialComponentContext(null);

  // initial serviceManager
  XMultiComponentFactory xLocalServiceManager =
xLocalContext.getServiceManager();

  // create a URL resolver
  Object urlResolver = xLocalServiceManager.createInstanceWithContext(
      "com.sun.star.bridge.UnoUrlResolver", xLocalContext);

  // query for the XUnoUrlResolver interface
  XUnoUrlResolver xUrlResolver =
      (XUnoUrlResolver) UnoRuntime.queryInterface(XUnoUrlResolver.class,
urlResolver);

  // Import the object
  Object rInitialObject = xUrlResolver.resolve(
    "uno:socket,host=localhost,port=2002;urp;StarOffice.ServiceManager");

// XComponentContext
if (null != rInitialObject) {
    System.out.println("initial object successfully retrieved");
} else {
    System.out.println("given initial-object name unknown at server side");
}
```

The usage of the `UnoUrlResolver` has certain disadvantages. You cannot:

- be notified when the bridge terminates for whatever reasons
- close the underlying interprocess connection
- offer a local object as an initial object to the remote process

These issues are addressed by the underlying API, which is explained in Opening a Connection.

Characteristics of the Interprocess Bridge

The whole bridge is *threadsafe* and allows multiple threads to execute remote calls. The dispatcher thread inside the bridge cannot block because it never executes calls. It instead passes the requests to worker threads.

- A *synchronous* call sends the request through the connection and lets the requesting thread wait for the reply. All calls that have a return value, an out parameter, or throw an exceptions other than a RuntimeException must be synchronous.
- An *asynchronous* (or `oneway`) call sends the request through the connection and immediately returns without waiting for a reply. It is currently specified at the IDL interface if a request is synchronous or asynchronous by using the [oneway] modifier.

Warning – Although there are no general problems with the specification and the implementation of the UNO `oneway` feature, there are several API remote usage scenarios where `oneway` calls cause deadlocks in OpenOffice.org. Therefore do not introduce new `oneway` methods with new OpenOffice.org UNO APIs.

For synchronous requests, *thread identity* is guaranteed. When process A calls process B, and process B calls process A, the same thread waiting in process A will take over the new request. This avoids deadlocks when the same mutex is locked again. For asynchronous requests, this is not possible because there is no thread waiting in process A. Such requests are executed in a new thread. The series of calls between two processes is guaranteed. If two asynchronous requests from process A are sent to process B, the second request waits until the first request is finished.

Although the remote bridge supports asynchronous calls, this feature is disabled by default. Every call is executed synchronously. The oneway flag of UNO interface methods is ignored. However, the bridge can be started in a mode that enables the oneway feature and thus executes calls flagged with the `[oneway]` modifier as asynchronous calls. To do this, the protocol part of the connection string on both sides of the remote bridge must be extended by `',Negotiate=0,ForceSynchronous=0'` . For example:

```
soffice "-accept=socket,host=0,port=2002;urp,Negotiate=0,ForceSynchronous=0;"
```

for starting the office and

```
"uno:socket,host=localhost,port=2002;urp,Negotiate=0,ForceSynchronous=0;StarOf
fice.ServiceManager"
```

as UNO URL for connecting to it.

Warning – The asynchronous mode can cause deadlocks in OpenOffice.org. It is recommended not to activate it if one side of the remote bridge is OpenOffice.org.

Opening a Connection

The method to import a UNO object using the `UnoUrlResolver` has drawbacks as described in the previous chapter. The layer below the `UnoUrlResolver` offers full flexibility in interprocess connection handling.

UNO interprocess bridges are established on the com.sun.star.connection.XConnection interface, which encapsulates a reliable bidirectional byte stream connection (such as a TCP/IP connection).

```
interface XConnection: com::sun::star::uno::XInterface
  {
      long read( [out] sequence < byte > aReadBytes , [in] long nBytesToRead )

           raises( com::sun::star::io::IOException );
      void write( [in] sequence < byte > aData )
           raises( com::sun::star::io::IOException );
      void flush( ) raises( com::sun::star::io::IOException );
      void close( ) raises( com::sun::star::io::IOException );
      string getDescription();
  };
```

There are different mechanisms to establish an interprocess connection. Most of these mechanisms follow a similar pattern. One process listens on a resource and waits for one or more processes to connect to this resource.

This pattern has been abstracted by the services com.sun.star.connection.Acceptor that exports the com.sun.star.connection.XAcceptor interface and com.sun.star.connection.Connector that exports the com.sun.star.connection.XConnector interface.

```
interface XAcceptor: com::sun::star::uno::XInterface
  {
      XConnection accept( [in] string sConnectionDescription )
           raises( AlreadyAcceptingException,
                   ConnectionSetupException,
                   com::sun::star::lang::IllegalArgumentException);

      void stopAccepting();
  };

  interface XConnector: com::sun::star::uno::XInterface
  {
      XConnection connect( [in] string sConnectionDescription )
           raises( NoConnectException,ConnectionSetupException );
  };
```

The acceptor service is used in the listening process while the connector service is used in the actively connecting service. The methods `accept()` and `connect()` get the connection string as a parameter. This is the connection part of the UNO URL (between *uno:* and *;urp*).

The connection string consists of a connection type followed by a comma separated list of name-value pairs. The following table shows the connection types that are supported by default.

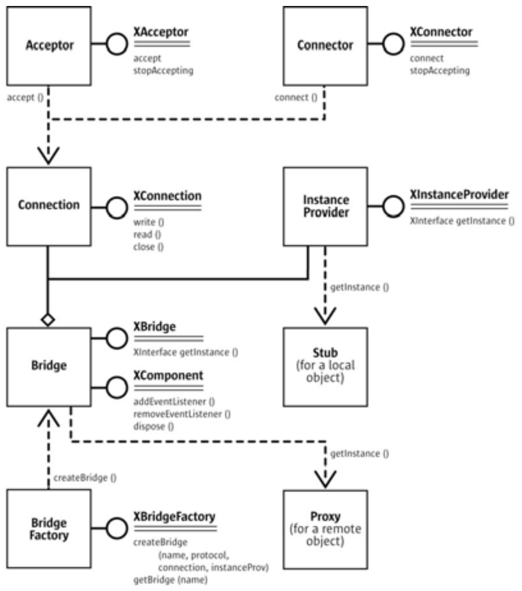
Connection type		
socket	Reliable TCP/IP socket connection	
	Parameter	**Description**
	host	Hostname or IP number of the resource to listen on/connect. May be localhost. In an acceptor string, this may be 0 ('host=0'), which means, that it accepts on all available network interfaces.
	port	TCP/IP port number to listen on/connect to.
	tcpNoDelay	Corresponds to the socket option tcpNoDelay. For a UNO connection, this parameter should be set to 1 (this is NOT the default - it must be added explicitly). If the default is used (0), it may come to 200 ms delays at certain call combinations.
pipe	A named pipe (uses shared memory). This type of interprocess connection is marginally faster than socket connections and works only if both processes are located on the same machine. It does not work on Java by default, because Java does not support	

named pipes directly

Parameter	Description
name	Name of the named pipe. Can only accept one process on name on one machine at a time.

Tip - You can add more kinds of interprocess connections by implementing connector and acceptor services, and choosing the service name by the scheme com.sun.star.connection.Connector.<connection-type>, where <connection-type> is the name of the new connection type.
If you implemented the service com.sun.star.connection.Connector.mytype, use the UnoUrlResolver with the URL `uno:mytype,param1=foo;urp;StarOffice.ServiceManager` to establish the interprocess connection to the office.

Creating the Bridge



The interaction of services that are needed to initiate a UNO interprocess bridge. The interfaces have been simplified.

The `XConnection` instance can now be used to establish a UNO interprocess bridge on top of the connection, regardless if the connection was established with a Connector or Acceptor service (or another method). To do this, you must instantiate the service com.sun.star.bridge.BridgeFactory. It supports the com.sun.star.bridge.XBridgeFactory interface.

```
interface XBridgeFactory: com::sun::star::uno::XInterface
  {
      XBridge createBridge(
              [in] string sName,
              [in] string sProtocol ,
              [in] com::sun::star::connection::XConnection aConnection ,
              [in] XInstanceProvider anInstanceProvider )
          raises ( BridgeExistsException ,
com::sun::star::lang::IllegalArgumentException );
      XBridge getBridge( [in] string sName );
      sequence < XBridge > getExistingBridges( );
  };
```

Note – In versions before OpenOffice.org 1.1, there are several differences.

The `BridgeFactory` service administrates all UNO interprocess connections. The `createBridge()` method creates a new bridge:

- You can give the bridge a distinct name with the sName argument. Later the bridge can be retrieved by using the `getBridge()` method with this name. This allows two independent code pieces to share the same interprocess bridge. If you call `createBridge()` with the name of an already working interprocess bridge, a `BridgeExistsException` is thrown. When you pass an empty string, you always create a new anonymous bridge, which can never be retrieved by `getBridge()` and which never throws a `BridgeExistsException`.
- The second parameter specifies the protocol to be used on the connection. Currently, only the 'urp' protocol is supported. In the UNO URL, this string is separated by two ';'. The urp string may be followed by a comma separated list of name-value pairs describing properties for the bridge protocol. The urp specification can be found on udk.openoffice.org.
- The third parameter is the `XConnection` interface as it was retrieved by Connector/Acceptor service.
- The fourth parameter is a UNO object, which supports the com.sun.star.bridge.XInstanceProvider interface. This parameter may be a null reference if you do not want to export a local object to the remote process.

```
interface XInstanceProvider: com::sun::star::uno::XInterface
  {
      com::sun::star::uno::XInterface getInstance( [in] string sInstanceName )

           raises ( com::sun::star::container::NoSuchElementException );
  };
```

The `BridgeFactory` returns a com.sun.star.bridge.XBridge interface.

```
interface XBridge: com::sun::star::uno::XInterface
  {
      XInterface getInstance( [in] string sInstanceName );
      string getName();
      string getDescription();
  };
```

The `XBridge.getInstance()` method retrieves an initial object from the remote counterpart. The local `XBridge.getInstance()` call arrives in the remote process as an `XInstanceProvider.getInstance()` call. The object returned can be controlled by the string `sInstanceName`. It completely depends on the implementation of `XInstanceProvider`, which object it returns.

The `XBridge` interface can be queried for a com.sun.star.lang.XComponent interface, that adds a com.sun.star.lang.XEventListener to the bridge. This listener will be terminated when the underlying connection closes (see above). You can also call `dispose()` on the `XComponent` interface explicitly, which closes the underlying connection and initiates the bridge shutdown procedure.

Closing a Connection

The closure of an interprocess connection can occur for the following reasons:

- The bridge is not used anymore. The interprocess bridge will close the connection when all the proxies to remote objects and all stubs to local objects have been released. This is the normal way for a remote bridge to destroy itself. The user of the interprocess bridge does not need to close the interprocess connection directly - it is done automatically. When one of the communicating processes is implemented in Java, the closure of a bridge is delayed to that point in time when the VM finalizes the last proxies/stubs. Therefore it is unspecified when the interprocess bridge will be closed.
- The interprocess bridge is directly disposed by calling its `dispose()` method.
- The remote counterpart process crashes.
- The connection fails. For example, failure may be due to a dialup internet connection going down.
- An error in marshaling/unmarshaling occurs due to a bug in the interprocess bridge implementation, or an IDL type is not available in one of the processes.

Except for the first reason, all other connection closures initiate an interprocess bridge shutdown procedure. All pending synchronous requests abort with a com.sun.star.lang.DisposedException, which is derived from the com.sun.star.uno.RuntimeException. Every call that is initiated on a disposed proxy throws a `DisposedException`. After all threads have left the bridge (there may be a synchronous call from the former remote counterpart in the process), the bridge explicitly releases all stubs to the original objects in the local process, which were previously held by the former remote counterpart. The bridge then notifies all registered listeners about the disposed state using com.sun.star.lang.XEventListener.

The example code for a connection-aware client below shows how to use this mechanism. The bridge itself is destroyed, after the last proxy has been released.

Unfortunately, the various listed error conditions are not distinguishable.

Example: A Connection Aware Client

The following example shows an advanced client which can be informed about the status of the remote bridge. A complete example for a simple client is given in First Steps.

The following Java example opens a small awt window containing the buttons **new writer** and **new calc** that opens a new document and a status label. It connects to a running office when a button is clicked for the first time. Therefore it uses the connector/bridge factory combination, and registers itself as an event listener at the interprocess bridge.

When the office is terminated, the disposing event is terminated, and the Java program sets the text in the status label to 'disconnected' and clears the office desktop reference. The next time a button is pressed, the program knows that it has to re-establish the connection.

The method `getComponentLoader()` retrieves the `XComponentLoader` reference on demand:

```
XComponentLoader _officeComponentLoader = null;

  // local component context
  XComponentContext _ctx;

  protected com.sun.star.frame.XComponentLoader getComponentLoader()
          throws com.sun.star.uno.Exception {
      XComponentLoader officeComponentLoader = _officeComponentLoader;

      if (officeComponentLoader == null) {
          // instantiate connector service
          Object x = _ctx.getServiceManager().createInstanceWithContext(
              "com.sun.star.connection.Connector", _ctx);

          XConnector xConnector = (XConnector)
UnoRuntime.queryInterface(XConnector.class, x);

          // helper function to parse the UNO URL into a string array
          String a[] = parseUnoUrl(_url);
          if (null == a) {
              throw new com.sun.star.uno.Exception("Couldn't parse UNO URL "+
_url);
```

```
          }

          // connect using the connection string part of the UNO URL only.
          XConnection connection = xConnector.connect(a[0]);

          x = _ctx.getServiceManager().createInstanceWithContext(
          "com.sun.star.bridge.BridgeFactory", _ctx);

          XBridgeFactory xBridgeFactory = (XBridgeFactory)
UnoRuntime.queryInterface(
              XBridgeFactory.class , x);

          // create a nameless bridge with no instance provider
          // using the middle part of the UNO URL
          XBridge bridge = xBridgeFactory.createBridge("" , a[1] ,
connection , null);

          // query for the XComponent interface and add this as event listener
          XComponent xComponent = (XComponent) UnoRuntime.queryInterface(
              XComponent.class, bridge);
          xComponent.addEventListener(this);

          // get the remote instance
          x = bridge.getInstance(a[2]);

          // Did the remote server export this object ?
          if (null == x) {
              throw new com.sun.star.uno.Exception(
                  "Server didn't provide an instance for" + a[2], null);
          }

          // Query the initial object for its main factory interface
          XMultiComponentFactory xOfficeMultiComponentFactory =
(XMultiComponentFactory)
              UnoRuntime.queryInterface(XMultiComponentFactory.class, x);

          // retrieve the component context (it's not yet exported from the
office)
          // Query for the XPropertySet interface.
          XPropertySet xProperySet = (XPropertySet)
              UnoRuntime.queryInterface(XPropertySet.class,
xOfficeMultiComponentFactory);

          // Get the default context from the office server.
          Object oDefaultContext =
              xProperySet.getPropertyValue("DefaultContext");

          // Query for the interface XComponentContext.
          XComponentContext xOfficeComponentContext =
              (XComponentContext) UnoRuntime.queryInterface(
                  XComponentContext.class, oDefaultContext);


          // now create the desktop service
          // NOTE: use the office component context here !
          Object oDesktop =
xOfficeMultiComponentFactory.createInstanceWithContext(
              "com.sun.star.frame.Desktop", xOfficeComponentContext);
          officeComponentLoader = (XComponentLoader)
              UnoRuntime.queryInterface( XComponentLoader.class, oDesktop);

          if (officeComponentLoader == null) {
              throw new com.sun.star.uno.Exception(
                  "Couldn't instantiate com.sun.star.frame.Desktop" , null);
          }
          _officeComponentLoader = officeComponentLoader;
      }
      return officeComponentLoader;
  }
```

This is the button event handler:

```
public void actionPerformed(ActionEvent event) {
      try {
          String sUrl;
          if (event.getSource() == _btnWriter) {
              sUrl = "private:factory/swriter";
          } else {
              sUrl = "private:factory/scalc";
          }
          getComponentLoader().loadComponentFromURL(
              sUrl, "_blank", 0,new com.sun.star.beans.PropertyValue[0]);
          _txtLabel.setText("connected");
      } catch (com.sun.star.connection.NoConnectException exc) {
          _txtLabel.setText(exc.getMessage());
      } catch (com.sun.star.uno.Exception exc) {
          _txtLabel.setText(exc.getMessage());
          exc.printStackTrace();
          throw new java.lang.RuntimeException(exc.getMessage());
      }
  }
```

And the disposing handler clears the `_officeComponentLoader` reference:

```
public void disposing(com.sun.star.lang.EventObject event) {
      // remote bridge has gone down, because the office crashed or was
terminated.
      _officeComponentLoader = null;
      _txtLabel.setText("disconnected");
  }
```

Service Manager and Component Context

This chapter discusses the root object for connections to OpenOffice.org (and to any UNO application) – the service manager. The root object serves as the entry point for every UNO application and is passed to every UNO component during instantiation.

Two different concepts to get the root object currently exist. StarOffice6.0 and