Request brokerage and remoting for OpenACS

Stefan Sobernig

July 2007

Contents

1 Introduction

xorb, the X(OTcl) R(equest) B(roker), is an infrastructure package for the web development toolkit OpenACS and OpenACS-based frameworks that provides for generic means of call abstraction. Call abstraction, hereby, refers to both distributed and non-distributed scenarios. In a non-distributed scenario, xorb is an object-oriented refinement of the well established OpenACS facilities also referred to as "service contracts'. In a distributed scenario, xorb provides a remoting infrastructure for OpenACS. xorb was designed in a protocol-agnostic manner, i.e. our primary intention is to provide support for a variety of remoting protocols. Protocol support is, therefore, realised in terms of protocol plug-ins for xorb. So far, we have realised a feature-rich SOAP protocol plug-in referred to as **xosoap**. Both, xorb and its protocol plug-ins are realised in XOTcl, a powerful OO extension to standard Tcl.

This guide, in its current shape, aims at providing the fundamentals of using call abstractions in a distributed scenario. We restrict ourselves, for the moment, to the remoting capabilities and therefore introduce the reader to the SOAP protocol plug-in, i.e. xotcl-soap (xosoap). The present document and resource collection will be steadily extended to cover more general aspects of request brokerage and protocol plug-ins available.

1.1 Sources of information

Beyond this manual, the following materials are currently available.

- A wiki version of this manual, allowing for commenting publicly and writing ahead.
- We aim at documenting elements of the public interfaces by means of the OpenACS in-code documentation facility and the API Browser. References will be given throughout the manual where appropriate. For inline documentation, watch out the OpenACS API Browser for xosoap's and xorb's interface.

- Tutorial prepared for the OpenACS Spring Conference 2007 (up-to-date):
  - Slide set
  - A podcast recording of my talk
- Tutorial (slide set) prepared for the OpenACS Fall Conference 2006 (obsolete)

Please, note that these resources reflect and document various stages of development and don't necessarily reflect the current state. The only source of information that is kept up-to-date is this manual document and its wiki mirror. Moreover, some of them might focus advanced concepts that are properly elaborated in the realm of this general-purpose documentation.

Besides, we assume some familiarity with basic XOTcl concepts and its syntax. There is an increasing number of resources available on this nifty language, you might want to check out the following resources to get started:

- There is a comprehensive XOTcl manual available.
- A reading tailored to OpenACS developers is to be found in XOTcl for OpenACS Developers. There is also a podcast on this topic available.

1.2 Copyright terms

2 Installation & configuration guideline

2.1 Prerequisites

2.1.1 Dependencies

- XOTcl module: We require XOTcl in version 1.5.4 (or higher) installed.
- xotcl-core: xorb/ xosoap are built upon version 0.51 (or higher).
- acs-service-contract: xorb/ xosoap use the abstraction layer for stored procedures as it comes with recent versions of xotcl-core (::xo:: db::sql::*). Since version 5.4.0d1, postgresql-based stored procedures defined for acs-service-contract are registered accordingly (acs_function_args table) and therefore exposed through the abstraction layer. However, having installed 5.4.0d1 is not mandatory, as we provide for a generic upgrade to lower versions of the package. This upgrade is non-invasive and non-critical to the overall functioning of your OpenACS installation.

2.1.2 Patches

As for the current release version (0.4), no additional patching is needed.

2.2 The XOTcl Request Broker (xorb)

2.2.1 Installation

1. Get and install the APM package: You may grab the trunk or tag version directly from svn, by calling either

```
svn <export | co> http://svn.thinkersfoot.net/xotcl-request-broker/
    trunk xotcl-request-broker
```

   or

```
svn <export | co> http://svn.thinkersfoot.net/xotcl-request-broker/
    tags/release-<version> xotcl-request-broker
```

   Here, we assume that you are in your packages directory. Alternatively, you might want to browse the current svn repository to get a tarball. Besides, APM tarballs are provided at media.wu-wien.ac.at

2. Proceed with the common way of installing OpenACS packages. Note, a restart of the server after the APM installation is recommended.

3. As a first step, you might want to point your browser to /request-broker/admin which features a first administrative cockpit for xorb.

2.2.2 Configuration

Configuration, primarily, refers to adjusting package parameters. Currently, there are no critical adjustments necessary at this level. In the future, we will provide necessary hints at this point.

2.3 The SOAP protocol plug-in (xosoap)

2.3.1 Installation

1. As with the request broker package, grab the APM package: Again, You may take trunk or tag version directly from svn, this time by calling either

```
svn <export | co> http://svn.thinkersfoot.net/xosoap/trunk xotcl-soap
```

   or

```
svn <export | co> http://svn.thinkersfoot.net/xosoap/tags/release-<
    version> xotcl-soap
```

Here, we assume that you are in your packages directory. Alternatively, you might want to browse the current svn repository to get a tarball. Besides, APM tarballs are provided at media.wu-wien.ac.at

2. Proceed with the common way of installing OpenACS packages. Note, a restart of the server after the APM installation is recommended.

3. Go and check out /xosoap/services/ where you might find some useful pointers on listening (example) services.

2.3.2 Configuration

In contrast to xorb, xosoap comes with a few configuration options for you. The following table provides an overview and some rough comments where necessary:

Option	Description
marshaling_style	Currently, xotcl-soap provides two marshaling styles that are partly related to the family of WSDL specifications and invocation schemes depicted by this specification. You might choose between RPC/Encoded (::xosoap::RpcEncoded), RPC/Literal (::xosoap::RpcLiteral) or Document/Literal (::xosoap::DocumentLiteral), respectively. Currently, we default to ::xosoap::RpcLiteral.
service_segment	The parameter value specifies the uri segment that will prefix url endpoints of services, i.e. http://<base_url>/<package_key>/<service_segment</<object_identifier>. It defaults to "services".
interceptor_config	xosoap allows for injecting specific handlers that are capable of intercepting and mangling requests and responses. Currently, xosoap comes with two configuration of interceptors chains, i.e. "standard" and "extended". This feature is currently not properly documented and subject to substantial changes in the near future.

3 A quick start guide

3.1 xorb

3.2 xosoap

The aim of this quick start section is to provide you with all the critical information required to get you started with integrating SOAP in your OpenACS application in terms of consuming an existing SOAP service and providing your own as quick as possible. We provide ready-made and take-away code snippets and discuss the important steps in further detail. At some point, we might refer to further readings or more advanced concepts.

↪ By referring to "SOAP" throughout the quick start section, we, more precisely, take into consideration SOAP 1.1 [3]. Moreover, we exclusively refer to RPC as messaging or rather (de-)marshaling style. So, any

occurrence of "SOAP" should actually be read as "SOAP-RPC 1.1".

3.2.1 How to glue to ...?

Say, you want to realise a scenario as depicted in Figure 1, i.e. you want to use the simple echo functionality provided by a remote object or remote procedure through SOAP. In this example scenario, we take a remote SOAP service called EchoService as a given. It might realised either in xosoap itself or any other SOAP infrastructure framework, such as gSOAP, Apache Axis, .NET Remoting and the like. Further, we assume that EchoService features a call "echoFloat" that requires a single argument of type "float" (as defined as "built-in" simple type by the XML schema specification) as input and promises to return a value of the very same type. How do you call and consume this little example SOAP service by means of xosoap?

↪ As for data type declarations to be used on OpenACS Service Contracts or remoting protocols, there is a section in this manual dedicated to this issue (see Section 4.3). For the scope of this quick start section, all (data) types refer to what is known as "built-in" primitive types as defined by XML schema specification 1.0 [2]. In xorb, there is a distinction between data types tags you can use in specifying contracts and the type handler behind the scenes. The former, e.g. xsFloat in the below example, are called type codes, the actual handler (an XOTcl object) is an anything.

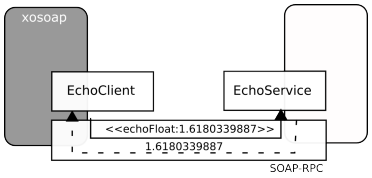

Figure 1: Our example scenario

↪ Hold on! If you prefer to skip the little code walk below, just look at the underlying and deployable example script xotcl-request-broker/www/doc/manual/examples/xosoap/example-01-soap-consumer.tcl or direct your browser to it to see it at work.

↪ Interested in more details on a few flavours for realising soap clients: 4.2.1

You just need four little steps and a few lines of code to get there:

1. Provide SOAP-specific configuration information as glue object. To be concrete, create an instance of the XOTcl class ::xosoap::SoapGlueObject and initiate the newly created object with some important bits of information:

```
1 set glueObject [SoapGlueObject new \
2                     -endpoint "http://websrv.cs.fsu.edu/~engelen/interop2.cgi
                          "\
3                     -action "http://websrv.cs.fsu.edu/~engelen/interop2.cgi" \
4                     -callNamespace http://soapinterop.org/ \
5                     -messageStyle ::xosoap::RpcEncoded]
```

In these four lines, we create and initiate an instance of the glue object class and provide three information bits to it. In line 2, we pass the actual endpoint address. In a conventional setting, the transport endpoint corresponds to a valid unique resource identifier (URL) as required by the HTTP protocol as transport provider. Line 4 also meets an information requirement at the HTTP level, i.e. the value assigned to a specific HTTP header field prescribed by SOAP 1.1 [3] (and 1.2) specifications: the SOAPAction header field. Line 2 and the setter call on callNamespace, on the contrary, refers to a higher protocol level, i.e. the SOAP level. The value of callNamespace will be used to set the namespace attached to the method-describing element. This is used by a few SOAP frameworks, e.g. .NET Remoting, to dispatch the remote call correctly to the responsible servant. For a detailed list of configuration options, please, refer to ::xosoap::SoapGlueObject.

Want to learn more ...

↪ ... about what 'glue objects' are? See 4.5.1

↪ ... about re-using 'glue object' to easen your tasks? See 4.5.2

2. Create a local proxy of the remote object or remote procedure, a so called client proxy. This client proxy object holds a proxy method for the remote method or procedure to call. There are a few flavours for specifying such a client proxy, the most straight forward, however, is by instantiating ::xorb::stub::ProxyObject and thus creating a proxy object.

```
6 ProxyObject EchoClient -glueobject $glueObject
```

The only significant step here is to associate the previously defined glue object to the client proxy. The information encapsulated by the glue object is then used by the underlying infrastructure to perform the actual remote call out of the combined information of the glue object and the client proxies interface. So, there it is, what we are still missing is the very client proxy.

3. Specify and realise the interface of your client proxy. The notion of object interfaces, to keep it simple, refers to the set methods and their method signatures defined on the object. When looking at Figure 1, we learn that the interface of the remote object EchoService is quite simplistic: There is a single method echoFloat that takes a single argument flagged "inputFloat". The argument is, apart from a concrete label, further qualified by a type constraint called xsFloat. These type constraints are realised as check options on non-positional arguments, in XOTcl terms. Besides, the interface promises a specific return value type, stipulated by passing a type code value as non-positional argument "returns" to ad_proc.

```
7  EchoClient ad_proc -returns xsFloat \
8      echoFloat {-inputFloat:xsFloat} {
9        By calling this proc, a remote call is issued \
10           against the previously defined endpoint
11     } {}
```

The method ad_proc that comes with ::xorb::ProxyObject overloads ad_proc as defined on ::xotcl::Object, see ::xorb::ProxyObject→ad_proc a detailed description.

4. Invoke the proxy method to perform the remote call

```
13 ns_write [EchoClient echoFloat -inputFloat 1.6180339887]
```

So far, it needed three declarative steps to realise a call to a SOAP service. The public interface of xosoap comes with various levels of granularity (see 4.2.1) with the above example referring to the medium-level interface. The higher-level one allows to realise the previous example in a total of three steps with only two declarative ones. Let's rewrite the example:

1. The major difference is that step 1 and 2 of the previous procedure are merged into one single step. Note, we now take advantage of a construct that natively comes with xosoap, the class ::xosoap::client::SoapObject. This makes it completely sufficient, for instance, to import from the Tcl namespace ::xosoap::client::* without any need to use facilities of xorb directly (see full code snippet of the following example in Listing 2.)

```
1  SoapObject EchoClient\
2      -endpoint "http://websrv.cs.fsu.edu/~engelen/interop2.cgi"\
3      -action "http://websrv.cs.fsu.edu/~engelen/interop2.cgi" \
4      -callNamespace http://soapinterop.org/ \
5      -messageStyle ::xosoap::RpcEncoded
```

2. The second step is, again, devoted to the declaration of the actual proxy interface, a local realisation of the echoFloat method on our combined proxy client and glue object:

```
6  EchoClient ad_proc -returns xsFloat\
7      echoFloat {
8        -inputFloat:xsFloat
9      } {
10       By calling this proc, a remote call is issued \
11           against the previously defined endpoint
12     } {}
```

3. Finally, once the above is done, you can continue by issuing the call, i.e. invoke on the "remote" method:

```
13 ns_write [EchoClient echoFloat -inputFloat 1.61805]
```

↪ See the complete example: Listing 2. All examples are dumped into the following directory, ready to pick them up: xotcl-soap/www/manual/examples.

↪ See (also for more example walk-throughs) Section 4.2.1

3.2.2 Getting glued to ...?

In this section, we will briefly look at some straight-forward ways to expose your OpenACS code as a SOAP-based service. For this very purpose, we, again, consider the scenario as introduced in the previous section, but this time, we shift focus. Now, we want xosoap to realise the EchoService, the callee, and not, as previously, the EchoClient, i.e. the caller. Figure 2 indicates that we now aim at realising a service that offers a single remote method,

i.e. "echoFloat", that takes a single argument of type "float" (again, as prescribed by the XML schema specfication) and is to echo the value of the argument back to the caller, again as XML schema built-in "float". The service laterally reverses the scenario in the previous section (see Section 3.2.1). How can you achieve this?

1. Declare and deploy an interface description. In OpenACS, interface descriptions have a long-standing tradition, however, sometimes neglected. You might be familiar with or, at least, you might have heart of service contracts as provided by the OpenACS core. If not, don't panic! For the time being, you just need to know that there is something called "service contract" in OpenACS, that xorb builds upon this core feature of OpenACS and that you now need to create such a service contract, weird enough. Think of an interface description as a fundamental sketch of an object interface that lays down how the interface of a remote object and the interfaces of client proxies mimicking this remote object are to be construed. As for the EchoService in Figure 2, a description might look the following way:

```
1  ServiceContract EchoService -defines {
2    Abstract echoFloat \
3        -arguments {
4          inputDate:xsFloat
5        } -returns returnValue:xsFloat \
6        -description {
7          Here, we outline an abstract call "echoFloat"
8          and its basic characteristics to be realised
9          both by servant code and client proxies.
10       }
11 } -ad_doc {
12   This contract describes the interface of the
13   EchoService example service as introduced by
14   xorb's manual.
15 }
```

   The above statement is, in the very literal sense of the word, a descriptive cast of a concrete EchoService service. Apart from a few bits of information targeting you as developer, it outlines that any servant or client proxy is meant to implement a method called "echoFloat", accepting a single argument and, in particular, throwing back a return value of a specific type. Don't forget to "deploy" your newly created service contract by calling:

```
16 EchoService deploy
```

   This separate step of deployment might seem an overhead in this simple example, but it renders useful when considering more complex scenarios of designing and lifecycling interface descriptions.

   ↪ There is a shrinking violet of an inline documentation available, see in particular, ::xorb::ServiceContract and ::xorb::Abstract.

2. Provide servant code and register it with the invocation mechanism. The lines above are, as the notion of "interface description" implies, are mere description of something.

What about the realisation of that something, i.e. the actual code block that is executed when the service is called. We refer to this code artefact as servant code or servant, in short. But simply creating servant code is not enough, you still have to let xorb know which code block to invoke when calls from client proxies occur. The latter is referred to as providing a "service implementation", i.e. registering your code piece as realisation of a specific service contract! Behind the scenes and in some use cases, it might be more appropriate to deal with these two steps separately, however, in a straight-forward "get-me-a-soap-service" scenario, xorb provides a short cut to accomplish both in a single step:

```
17  ServiceImplementation EchoServiceImpl \
18      -implements EchoService \
19      -using {
20        Method echoFloat {
21          -inputFloat:required
22        } {Echoes an incoming float} {
23          return $inputFloat
24        }
25      }
```

The non-positional argument -implements identifies the service contract realised by the underlying servant code, while -using allows for specifying the intended delegations, which concrete code block is meant to be called for which abstract call. In our concrete example, we both provide a servant method called echoFloat and, behind the scenes, this servant method is registered as callee for abstract calls on echoFloat as defined by the contract. Note that the facility ::xorb::Method is actually an XOTcl object (i.e. a slot object, to be more precise) that mimics the declaration of instprocs and procs as known from XOTcl. Finally, also deploy your service implementation:

```
26  EchoServiceImpl deploy
```

At this point, you have accomplished most of the work needed to provide your EchoService. Two things remain to be done:

↪ Interested in some background reading on OpenACS's service contracts? See the Section 5.1.

↪ See the complete Listing 8.1.

↪ Again, you might also want to watch out the API Browser for information on ::xorb::ServiceImplementation, ::xorb::Delegate, and ::xorb::Method.

3. You might have asked yourself, while going through step 1 and 2, where to actually put your the code outline above. Well, there are several answers to this question, you may choose between the following options:

   - Your package's library files (*-procs.tcl, *-init.tcl): Preferably, drop your code in files situated in the tcl-subdirectory of your package. In either case, whether you choose a *-procs or a *-init file, make sure that you place the following line before the actual code block depicted in step 1 and 2: `::xo::db::package require xotcl-request-broker`. Besides, there is a subtle difference between the *-procs and *-init files. The latter are evaluated only once, during server start-up. This means,

contracts and implementations are only evaluate once. If you constantly develop, let's say, the servant code declared by your implementation, and you want your modifications/ improvements to take effect without a dedicated server re-start, go for the *-procs option. This way, you can take advantage of xorb's reload and watch features.

- Your package's W(eb) U(user) I(nterface) files: In principle, you can specify a contract (::xorb::ServiceContract) in one of your www-subdirectory files as well. While it limits their usage, it does not break any functionality. Implementations (::xorb::ServiceImplementation) might as well be specified in WUI files, however, only the most basic variant makes sense in this context. The code of step 2 would not make sense as it defines servant code as well which should be available for all connections, i.e. in the context of all connection threads, and not only the one you specified it in by calling the WUI file.
- Developer shell: You can also drop the code in the shell as provided by the Developer Support package, however, the limitations for WUI scripts apply here as well.

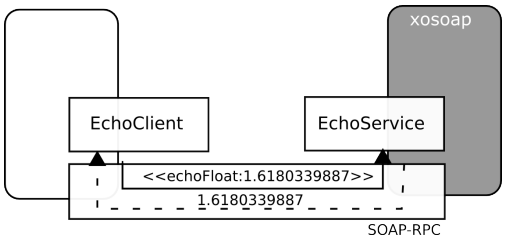

Figure 2: Our example scenario – The provider side

Finally, we want to look at two further nifty features for writing the callee interface, i.e. the service implementation, and servants. The first one refers to what is known as ::xorb::Delegate, yet another way of linking abstract definitions to concrete servant code blocks. First, the above example could be rewritten in a probably common scenario. Just imagine, you previously created a piece of code (e.g., a proc) and you simply want to expose this artefact as servant instead of creating one by using ::xorb::Method as shown above. Or, you simply articulate the requirement that you keep prospective servant code and mere registration or binding code neatly seperated. In either case, you can revert to the use of little forwarding feature, i.e. ::xorb::Delegate. Let's rewrite Listing 8.1 accordingly:

The starting point is the assumption that there is an existing code block that we want taking the role of a servant for the abstract echoFloat method defined by our example contract. This particular proc might take the following form:

```
1 namespace eval :some_name_space {
2   proc servantProc {inputFloat} {
3     return $inputFloat
```

```
4     }
5   }
```

In view of our code base, we could then rewrite our service implementation the following way, using ::xorb::Delegate.

```
6   ServiceImplementation EchoServiceImpl \
7       -implements EchoService \
8       -using {
9         Delegate echoFloat \
10            -per-object true \
11            -private_p true \
12            -for ::some_name_space::servantProc \
13            {Echoes an incoming float}
14      }
```

Let's, for a second, neglect some minor varations we introduced in the above lines and that might distract. Key to the example is that the service implementation now, once the specification has been successfully processed, won't host any callable method on its own, it rather acts as forwarder to your candidate proc. This delegation step is realised by providing an absolute and qualified reference to the target proc to the non-positional argument -for. Apart from the changed semantics, it will produce the same result as Listing 8.1. Some refinements are also shown in the above snippet that did not show up in the introductory listing. You might have spotted that ::xorb::Delegate takes a couple of extra parameters. The same set is available for ::xorb::Method. In fact, both support all parameters known from ad_proc (or ad_instproc). In addition, they come with an optional parameter -per-object. If you are familiar with XOTcl, you will have noticed that methods, and beyond, can be declared directly on objects or for instances of the declaring object. Therefore, the distinction between "proc" and "instproc", just to name the most prominent one. Similarily, you define the ::xorb::Delegate and ::xorb::Method either at the per-object or per-instance level. Behind the scenes, xorb's invocation mechanism detects either way and will handle dispatches accordingly.

↪ You might want to check out the API Browser for ::xorb::Delegate and ::xorb::Method.

As you just learnt, ::xorb::Delegate provides for rather simple delegating dispatch of abstract calls. A thorough look at the above example of a serving proc, i.e. "servantProc", might have created some suspicion: Tcl procs allow for positional argument passing, if we neglect the ad_proc facility of OpenACS for a second. However, xorb is primarily built around non-positional arguments. We briefly mentioned this fact while describing the semantics of ::xorb::Abstract. This is due to the fact that xorb heavily uses XOTcl check options that are (at least up to the 1.5.x generation) exclusively available to non-positional arguments. Besides, most remoting protocols, including SOAP and XML-RPC adopted this pattern of argument passing [9].

Be that as it may, it leaves xorb with servants that do not take non-positional arguments. However, thanks to (XO)Tcl's introspective capabilities, xorb provides for argument bridging. It will identify the type of servant code block and adapt the pending call dispatch accordingly, including the rewrite of the passed argument-value list. Note, there are of course limitations to this approach, namely if you consider mixed signatures, containing both positional and non-positional arguments.

4 Advanced usage

4.1 How to test your services

4.2 Public interface

4.2.1 What is the structure of xorb's and xosoap's public interfaces?

As can be seen from Figure 3, the interface available is organised in three packages and at the same time levels of granularity. The lowest-level interface is deeply integrated with XOTcl constructs, especially ::xotcl::Object and ::xotcl::Class. xorb, therefore, allows to turn any XOTcl object into a client proxy to take advantage of call abstractions (either in a distributed or non- distributed scenario). At this level, xorb introduces two particular keywords (i.e. methods) that allow to define proxy methods in the literal sense of the word on ::xotcl::Object and its sub classes, ::xotcl::Object→glue (or ::xotcl::Object→ad_glue). These methods appear as keywords (one might refer to modifiers, though this is not appropriate strictly speaking) to ordinary proc or instproc declarations, however, these method declaration don't require or allow method bodies to be defined. The example provided in the quickstart guide (see 3.2 can be realised with any XOTcl object, by simply writing:

1. First, again, create a "glue" object and pass the necessary call information:

```
1  set glueObject [SoapGlueObject new \
2                      -endpoint "http://websrv.cs.fsu.edu/~engelen/interop2.cgi"
                           \
3                      -action "http://websrv.cs.fsu.edu/~engelen/interop2.cgi" \
4                      -callNamespace http://soapinterop.org/ \
5                      -messageStyle ::xosoap::RpcEncoded]
```

2. In this step, you can simply revert to the use of an ordinary object. Either create one (as shown in the example below) or use existing objects and make them proxy objects by, first, attaching the glue object to it and, second, declare a proxy interface by means of ::xotcl::Object→ad_glue (or ::xotcl::Object→glue):

```
6  ::xotcl::Object EchoClient -glueobject $glueObject
```

```
7  EchoClient ad_glue -returns xsFloat\
8      proc echoFloat {
9        -inputFloat:xsFloat,glue
10     } {
11       By calling this proc, a remote call is issued \
12           against the previously defined endpoint
13     }
```

3. The actual call is effectuated the same way as in the previous example:

```
14 ns_write [EchoClient echoFloat -inputFloat 1.61805]
```

At this level, proxy methods appear in their purest forms as mere placeholders for abstract calls realising a local representation of a remote (object) interface. Please, note that declarations by means of ::xotcl::Object→glue/ ::xotcl::Object→ad_glue don't take a method body as they are mere interface descriptors. This makes them syntactically comparable to the abstract keyword that comes with XOTcl to mimic abstract classes. However, do not align them at the semantic level.

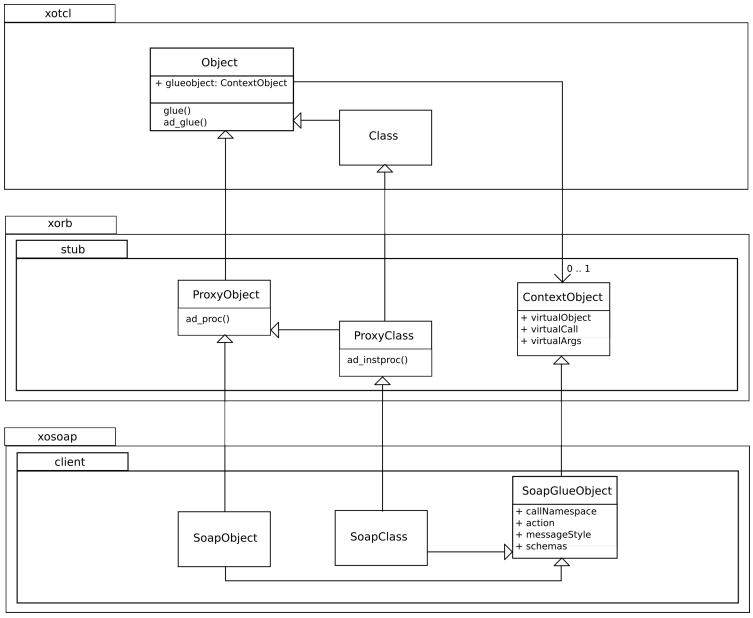

Figure 3: Class diagram of interface entities

The medium-level interface is already familiar to you when reviewing the quick start section. It is provided by xorb and more or less introduces two xorb-specific constructs, i.e. ::xorb::stub::ProxyObject and ::xorb::stub::ProxyClass. While they inherit all capabilities of the low- level interface, they provide additional facilities to create and use proxy objects. A glimpse at Figure 3 reveals that they overwrite the methods ad_proc provided by ::xotcl::Object and ad_instproc offered by ::xotcl::Class. This already suggests that proxy interfaces can now be declared in a slightly different manner than by ::xotcl::Object→glue/ ::xotcl::Object→ad_glue. While you can still revert to the former, ::xorb::stub::ProxyObject→ad_proc and ::xorb::stub::ProxyClass→ad_instproc allow for interface descriptors with method bodies. When looking at Listing 2, you might notice that the argument declaring the method body is empty (corresponds to an empty Tcl string). For the sake of simplicity, we did not introduce a notable feature of the medium-level interface at this stage that we refer to as proxy templates. For a primer on this concept and examples see Section 4.4.

The high-level interface, as can been seen from Figure 3, is a purely protocol plug-in-specific one. As for xosoap, we provide two intregating constructs: ::xosoap::client::SoapObject

and ::xosoap::client::SoapClass inherit, expose, and combine properties of glue objects and proxy clients in one. This allows for an abbreviated and compact writing of SOAP client code, with all facilities introduced by lower interface levels. For an introductory example see the quick start section on xosoap (see Section 3.2) and, in particular, Listing 2.

4.3 Data type infrastructure

Providing framework support for data type handling, especially in a cross-language and multi-protocol setup is a challenging task. Nevertheless, framework designers and developers may revert to well documented experiences on these issues and do not have to start from scratch. By these documented experiences, we primarily refer to a set of design patterns, some of them to be found in [9, 7, 6]. For the very moment, we do not want to go into details of the framework implementation, rather provide a quick overview on, first, terminology and, second, facilities to be used by you, the developer.

As for terminology, there are just very few concepts to remember. If you recall the examples in specifying client proxies or service contracts (see Section 3.2), the way you specified the signature of a proxy method or ::xorb::Abstract always involved specific check options. These might look familar:

```
    echoFloat {-inputFloat:xsFloat} {
```

```
    Abstract echoFloat \
        -arguments {
          inputDate:xsFloat
        } -returns returnValue:xsFloat \
```

Check options, again, are a native XOTcl concept to formulate constraints of various types on non-positional arguments. In xorb, however, we use this native XOTcl mechanism to specify special, i.e. protocol-specific type constraints. "xsFloat", as used and shown in the above examples, is referred to as type code, i.e. a short-cut identifier for a specific type handler. The actual type handler, i.e. ::xosoap::xsd::Float is named anything type, or simply anything. The main idea of anythings are nicely elaborated on by [7, 6], however, the main idea is to provide a generic and uniform value container that comes with on-demand casting.

↪ Note, some clarifying side notes might be required at this point: The term casting may not be mistaken for what it refers to in compiled and (strongly) typed languages. In the realm of xorb, it simply refers to exposing a value in accordance with some value space and representation constraints.

↪ Another side note might be appropriate. The term "type code" is prominently used in another, pure-SOAP framework written in and for Python, i.e. the Zolera SOAP infrastructure (ZSI, [1]). However, type codes though realising a comparable aim, are properties to Python objects later used by type-specific (de-)marshallers. In xorb, type code serve a more declarative purpose, simply by hijacking the check option mechanism of XOTcl.

The third and last concept realises what is need to provide for multi-protocol support. Just imagine, you want to re-use, or rather, expose a service contract originally defined for the local use in an OpenACS instance as interface description for a SOAP-based service. Type codes such as string, integer, etc. need to be resolved for the scope of SOAP and WSDL handling. Therefore, xorb comes with the idea of type sponsorships. A type sponsor is

a data type handler, i.e. an anything, that comes with a protocol plug-in and announces, upon initialisation, itself as sponsor to a xorb general-purpose anything. The xorb-specific anything then, upon being processed during the handling of invocation dispatches, resolves its sponsor for the actually used protocol.

Find below a list of currently supported types, providing the type codes at hand and the anythings acting in the background. Rows read as sponsorship relationships (if realised).

xorb types (type code / anything)	xosoap types (type code / anything)	comments
void | ::xorb::datatypes::Void	xsVoid | ::xosoap::xsd::XsVoid	Void is a particular type descriptor (not featured in XML schema or the like), more or less, making explicit when block calls do not convey results in explicit invocation. This might not be unambiguously expressible by language means, as in Tcl, and is a particular challenge in cross-language settings.
string | ::xorb::datatypes::String	xsString | ::xosoap::xsd::XsString	See xsd:string
integer | ::xorb::datatypes::Integer	xsInt | ::xosoap::xsd::XsInt	See xsd:int. Note, XOTcl actually comes with a native "integer" check option, this is, however, overdefined in the realm of xorb. The sponsorship of xsd:int for a Tcl integer is not yet quite decided, but it has been proven useful in known scenarios.
boolean | ::xorb::datatypes::Boolean	xsBoolean | ::xosoap::xsd::XsBoolean	See xsd:boolean
timestamp | ::xorb::datatypes::Timestamp	n/a	We assumed, at the xorb side, a correspondence to the SQL:1999 data type "timestamp" WITHOUT timezone
uri | ::xorb::datatypes::Uri	n/a	We provide for some RFC 3986 validation.
version | ::xorb::datatypes::Version	n/a	Uses regular expression taken from OpenACS package manager.
float | ::xorb::datatypes::Float bytearray | ::xorb::datatypes::Bytearray	xsFloat | ::xosoap::xsd::XsFloat n/a	See xsd:bytearray At the xorb-side, it is not clear to us what is actually labelled "bytearray" and which particular forms of Tcl strings are meant to be ruled by this type description.
object | ::xorb::datatypes::Object	soapStruct | ::xosoap::xsd::soapStruct	This sponsor relationship might change in the near future, as soon as the marshaling styles in xosoap are consolidated. The use of ::xotcl::Objects in xorb is not quite clarified, but shown here for the sake of completeness.

n/a	xsDecimal | ::xosoap::xsd::XsDecimal	See xsd:decimal
n/a	xsInteger | ::xosoap::xsd::XsInteger	See xsd:integer
n/a	xsLong | ::xosoap::xsd::XsLong	See xsd:long
n/a	xsDouble | ::xosoap::xsd::XsDouble	See xsd:double
n/a	xsDate | ::xosoap::xsd::XsDate	See xsd:date
n/a	xsTime | ::xosoap::xsd::XsTime	See xsd:time
n/a	xsDateTime | ::xosoap::xsd::XsDateTime	See xsd:dateTime
n/a	xsBase64Binary | ::xosoap::xsd::XsBase64Binary	See xsd:base64Binary
n/a	xsHexBinary | ::xosoap::xsd::XsHexBinary	See xsd:hexBinary
n/a	soapArray | ::xosoap::xsd::SoapArray	Depending on the message/marshaling style we either go for the SOAP encoded array or the WS-I compliant array notation.

4.4 Proxy templates

Proxy templates or rather template methods on proxy objects follow a primary intention: Extending the use of pure interface descriptors to full-featured methods that allow to pack program logic along with a call abstraction in a single method body. Our intention can be best summarised when following the idea of the Template Method pattern as documented by [4]. As the term "template" suggests, there might be scenarios that integrate call abstraction, or in our remoting scenario simply remote calls, in more complex (i.e. multi-step) behaviour. This is, certainly, also realisable with the lower-level interface. However, it would require another construct, either a proc, or a sub class etc., that combines the general behaviour and the invocation references to the abstract calls. By means of proxy template methods as introduced by the medium-level interface you can know write template code and invocations of abstract calls in the same method block, making encapsulation in this respect more seamless. A primary, and still easily graspable example are conditional invocations of abstract calls. Let's take the following example of Listing 1 where we simply rewrite lines 16 – 21. This yields the following (see Listing 4 for the complete example):

```
1 EchoClient ad_proc -returns xsFloat \
2     echoFloat {-inputFloat:xsFloat,glue} \
3     {
4       By calling this proc, a remote call is issued \
5           against the previously defined endpoint. \
6           However, this time, the call only gets executed \
```

```
7           when the floating number matches a certain value \
8           space.
9       } {
10        if {[expr {round(0.5 * (1 + sqrt(5)))}] == [expr round($inputFloat)]} {
11          next;# invoke on remote method / procedure
12        }
13      }
```

We draw your attention to the last argument of the ::xorb::stub::ProxyObject→ad_proc call which perfectly resembles the declaration of a method body. The exemplary condition we introduce here is only to simplistic but delivers the message. You might have noticed that the floating point number used in the EchoClient/ EchoService example corresponds to the decimal representation of the golden ratio phi. We, now, in our proxy template method, introduce a constraint which limits the value space accepted to all decimals that round to the same integer as the most accurate decimal expansion of the golden ratio, given by the formula $(1 + \sqrt{5})/2$. If, and only if this value validation is passed, we want the abstract call to be invoked. At this point, you might have realised that the ::xotcl::next command is the forwarder to the actual proxy method.

Now, once you got the main idea behind this example, take a closer look at the declaration of arguments in line 17. You might notice that there is a slight but important modification of how the argument inputFloat is specified. In fact, we added an additional check option "glue" which is mandatory as soon as you provide a non-empty method body to either ::xorb::stub::ProxyObject→ad_proc and ::xorb::stub::ProxyClass→ad_instproc. There are two reasons for this requirement: First, regarding concepts and clarity for you as developer, you are now specifying two method records (i.e. set of arguments plus check options) within one single step. On the one hand, the record for the decorating method (the proxy template method) called "echoFloat" and, on the other hand, the actual proxy method "echoFloat". We, therefore, distinguish between an inner and and outer record. Both records might be identical but not necessarily. Second, regarding internals, it is required to explicitly distinguish between elements of either record. Lets have a look at another variation of the previous example (see Listing 5):

```
1   EchoClient ad_proc -returns xsFloat \
2       echoFloat {
3         -inputFloat:xsFloat,glue
4         -nonpositionalArgument
5         positionalArgument1
6         positionalArgument2
7       } \
8       {
9         By calling this proc, a remote call is issued \
10            against the previously defined endpoint. \
11            However, this time, the call only gets executed \
12            when the floating number matches a certain value \
13            space.
14      } {
15        ns_write "Variables in this scope: [info vars]\n"
16        if {[expr {round(0.5 * (1 + sqrt(5)))}] == [expr round($inputFloat)]} {
17          next;# invoke on remote method / procedure
18        }
19      }
```

Referring to lines 3, we extended the method's record by an additional non-positional argument and two positional ones. In fact, "annotating" a non-positional argument with "glue" assigns it to be member in the proxy record, while all the others, especially the other non-positional arguments, won't be interpreted as elements of the proxy interface. At the same time, you will be able to use equally named variables containing the arguments" values in the scope of the proxy template method. The final call to the proxy object by means of the template method might take the following form:

```
20 ns_write [EchoClient echoFloat \
21              -nonpositionalArgument "npValue" \
22              -inputFloat 1.6180339887 \
23              "val1" "val2"]
```

Apart from the constraint scenario, template methods also proof useful when you (for whatever reason) want to deviate from the proxy interface (i.e. the inner record). This might true in the following cases:

- In case you need to mangle the argument values before being passed to the remote method, or
- if you do not expose all arguments stipulated by the proxy interface (i.e. the inner record) in the outer record.

In the following variation of an initial listing (see Listing 4), we give provide an example for the first case. Let" see:

```
1     } {
2       ns_write "Variables in this scope: [info vars]\n"
3       set inputFloat [expr "$inputFloat + 1"]
4       next -inputFloat $inputFloat;# invoke on remote method / procedure
5     }
```

Here, we simply take the inbound value for inputFloat and increase it by 1. This updated value should then be passed to the remote method to be communicated to the call point. Two important aspects should be retained from this snippet: First, if you do so, you have to explicitly append argument-value pairs to ::xotcl::next. Otherwise, ::xotcl::next, if called without any appended terms, passes on the last argument list on the call stack. Second, if you decide to do so, you have to provide all arguments to ::xotcl::next that are required by the method shadowed by ::xotcl::next. Required arguments are both non-positional arguments flagged "glue" or "required" and positional arguments with no default values in the record declaration.

↪ We assume that your are familiar with the basic mechanism of inheritance XOTcl delivers to you. If you are new to XOTcl or its conceptual "sponsors" such as CLOS etc., you might want to consider studying the relevant sections.

↪ You might want to give it a try. Just edit the last line of the example outlined above and change the value passed as "inputFloat" so that it once validates correctly and once fails.

Using ::xotcl::next in this setting is both a slight redefinition of its semantics in view of XOTcl inheritance, and a pointer to the internal implementation of our proxy templates or proxy template methods: The declarative information passed to ad_proc or ad_instproc is used both (a) to specify a proxy method and register it with invocation infrastructure and (b) to create a decorator to the actual proxy object in terms of a XOTcl mixin class. This decorator hosts the method body as declared with ::xotcl::next resolving to the shadowed proxy method.

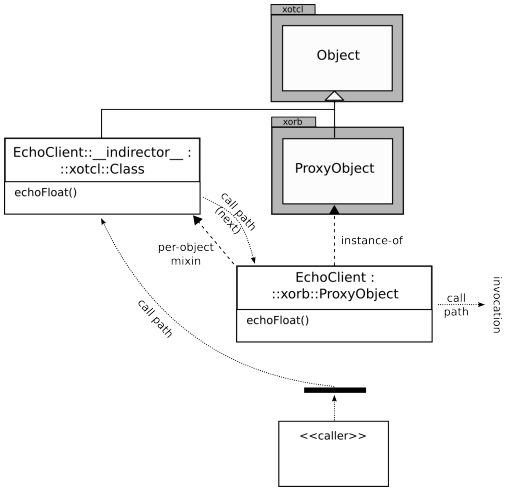

Figure 4: Call indirection as used for proxy templates

Figure 4 aims at sketching the internals of proxy template methods and the behaviour shown above. Again, we assume some familiarity with XOTcl concepts, especially mixins as XOTcl's realisation of abstract sub classes. The main messages from Figure 4 are the following:

- Calls upon objects that mixin classes are registered with are indirected according to a certain order of precedence, visualised by the call path in Figure 4.
- What previously has been referred to as outer interface is the method record of echoFloat defined on the mixin class (i.e. :: EchoClient::__indirector__). The inner record refers to the record declaration of echoFloat of EchoClient.
- The original or native call path in XOTcl would continue along the inheritance and realisation (instantiation) path, involving e.g. ::xorb::stub::ProxyObject and Object in Figure 4. However, in the context of proxy templates, the call as such leads to the actual invocation of the abstract call.

↪ The subtle difference between inner (proxy) record and outer (template) record should be kept as important message from the previous section. It provides a powerful mechanism to the developer, however, might be a confusing way of mixing-up signatures at first glance.

4.5 Glue objects

4.5.1 What is the idea of glue objects?

Glue objects, as represented by all sub-classes of ::xorb::stub::ContextObject, are the xorb-specific flavour of what is more commonly known as context objects in language design. Context objects, in short, represent a specific pattern of argument passing applicable to various object-based language environments [9]. The technique comprises the use of a generic or specifically typed object as single argument to operations with the actual arguments being stored as per-object variables (or more commonly, instance variables). The use of context objects has been suggested for a set of scenarios. These might be reflected against the requirements and design decisions taken in xorb's proxy interface (see Section 4.2).

First, it allows to avoid the escalation of operation signatures which is a common risk when a considerable amount of arguments needs to be passed. This is certainly given in the scope of xorb and its protocol plug-ins, just to take ::xosoap::client::SoapGlueObject as an example. The list of properties is, by the way, not taxonomic but rather open for further extensions provided that this required due to a couple of reasons. The most prominent one is concatenation of protocol-specific shadow information, e.g. whenever a new transport handler (SMTP, JMS) is added it might require new properties at the level of the most specific glue object. Another source of an increased need of top-down configurability is the considerable level of heterogeneity between remoting frameworks interacting. This heterogeneity ranges from variation in (de-)marshaling styles, resolution mechanisms to fundamental modes of interaction.

Second, the quality of arguments, i.e. their varying structure, might be another source of complexity that can be better abstracted from by means of context objects. This is somehow closely related to aspect five outlined below.

Third, the set of arguments to be passed might vary from invocation to invocation, a requirement that is rather hard to serve by more conventional argument passing techniques in an elegant and effective manner. The actual argument being an object or even a typed object, we might revert to the given mechanisms of inheritance and polymorphism to model these variations more appropriately.

Fourth, escalated operations signatures are hard to maintain in generic framework designs, where an arbitrary number and arbitrary kinds of clients manipulate these bits of information. This is valid for the generic extension mechanisms that come with xorb, for instance the interceptor infrastructure.

Fifth, context objects allow to specify an interface or even an protocol for manipulating the stored argument data. By interface, we refer to a set of accessors, for instance, a protocol, however, would require more sophisticated features. Though this can be realised in various language settings, we now concentrate on XOTcl. XOTcl comes with a specific kind of objects, so called slots, that generalise the idea of object properties or parameters (as known from other environments) and introduce manager objects for object-level variables. Slots allow various fine-granular interceptions to be executed upon manipulation of object states. This allows to enforce protocols in terms of canonical naming, validation of value spaces etc. in a neat object-centric manner.

Context objects in the scope of remoting frameworks, in their role as invocation contexts, are thoroughly discussed and exemplified in [8]. A more general sketch in the same direction

but applicable to a wider range of domains can be found in [5].

Context objects are also applied as solution to another design problem in xorb, namely the provision of a generic infrastructure for data type handling across boundaries of protocol plug-ins (see Section 4.3). This relates mainly to another flavour of the concept of context objects elaborated on both by [6, 7].

4.5.2 How to use scoping on glue objects for re-use?

Against the basic notion of context objects (see Section 4.5.1), glue objects, the xorb-specific flavour of context objects, have already been introduced in at least two notational forms in the quick start guide (see 3.2) to the reader. There is, however, an advanced usage that can be applied to glue objects that we refer to as scoping. Behind the scenes, this feature is the result of a specific order of precedence when resolving the glue object that, in turn, is used for further processing of a call issued upon a client proxy. There are two dimensions of precedence, one regarding the moment of resolution (declaration or call time) and the scope of reach of a glue object (see Figure 6). As shown in Figure 5, upon issue of a call upon a

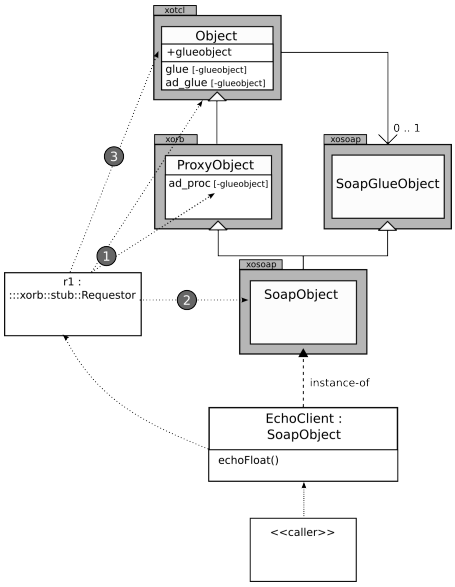

Figure 5: Precedence order for resolving glue objects

client proxy (EchoClient), the ::xorb::stub::Requestor follows a three-level precedence:

1. The glue object reference passed to the non-positional argument "glueobject" on either ::xotcl::Object→glue, ::xotcl::Object→ad_glue, ::xorb::stub::ProxyObject→ad_proc and ::xorb::ProxyClass→ad_instproc take the highest precedence. This order of precedence is enforced upon declaration time, i.e. the reference is compiled into the body of

the proxy method. This level of specifying a glue object allows to realise client proxies as mere collections of remote procedure proxies (and not proxies in the OO sense).

2. As can be learnt from Figure 5 and introduced in Listing 2, the role of client proxy and glue object are unified in a set of objects, namely ::xosoap::client::SoapObject or ::xosoap::client::SoapClass. If the requestor identifies a proxy object of this kind, it will be granted second highest precedence. We refer to this level as the per-object scope (see Figure 6). As for the scope, it is applicable to all proxy methods defined on the client proxy.

3. In Listing **??** we can find that there is also a property "glueobject" available for all objects of type ::xotcl::Object. This property, or rather the glue object assigned to it, takes the lowest precedence in the resolution order. Comparable to level 2, it applies to all proxy methods defined, however, as the glue object is a distinct entity from the client proxy, it is potentially re-usable for various client proxies. Therefore, we refer to the level of precedence as the shared scope (see Figure 6).

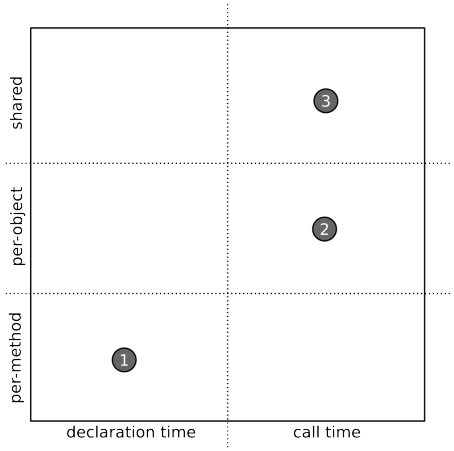

Figure 6: Dimensions of precedence: resolution time & scope of reach.

A comparison of these three configurations yields some insights on (dis-)advantages and provides some hands-on usage examples. Using both the per-method and per-object/ shared scope within a single proxy object allows to implement more than just one interface through a single proxy object. However, this decision needs to be taken upon declaration time as this is a limitation of the current per-method scoping. Also, we so provide the means to specify both proxies for remote methods (in the strict OO sense) and remote procedures in a single entity. Let's look at a slight modification of Listing 2 that aims at illustrating this first configuration. In a first step, we add a second glue object which contains a different set of configurations options, in particular it targets a different endpoint:

```
1  set glueObject [SoapGlueObject new \
2                      -endpoint "http://dietrich.ganx4.com/nusoap/testbed/
                           round2_base_server.php"\
3                      -messageStyle ::xosoap::RpcEncoded]
```

Note, EchoClient as declared in Listing 2 already contains shadow information of a glue object. If we now pass the second glue object defined to the ad_proc declaration, this second glue object takes precedence over the first one, i.e. EchoClient itself.

```
4  EchoClient ad_proc -glueobject $glueObject -returns xsFloat\
5      echoFloat {
6        -inputFloat:xsFloat
7      } {
8        By calling this proc, a remote call is issued \
9            against the previously defined endpoint
10     } {}
```

The ultimate result of this minor tweak in Listing 6 is that the call will be delivered to a remote object that listens at a different endpoint than the one in Listing 2.

Both, per-object and shared scope, allow for more simplified notations of proxy objects. As for re-use, the per-object scope is primarily centred around the re-use of the proxy interface as represented by the proxy object while continuously, over the program flow, adapting the configuration in terms of glue object manipulations. The shared scope puts emphasis on the re-use of a given glue object on multiple proxy objects. Moreover, it allows for dynamic, non-hierarchical refinements through mixins. In doing so, we gain the opportunity can inject glue objects for a selected period of time upon certain conditions into client proxies.

Again, to provide a hands-on experience, let's have a look at Listing 7. In fact, we try to achieve the same behaviour as realised in Listing 6 but by completely different means. This time we want to decorate our previously defined client proxy at an arbitrary moment during run time. For this purpose, we need an appropriate decorator, such as:

```
11 ::xotcl::Class ProxyDecorator
12 SoapGlueObject ProxyDecorator::GlueObject \
13     -endpoint "http://dietrich.ganx4.com/nusoap/testbed/round2_base_server.php"\
14     -messageStyle ::xosoap::RpcEncoded
15 ProxyDecorator instproc glueobject args {
16   return [self class]::GlueObject
17 }
```

The decorator, an XOTcl mixin class, can be realised in manifold ways, the above is only one of a couple of options at hand. The main idea, however, is that a new glue object (ProxyDecorator::GlueObject) is return upon calls to the decorator's "glueobject" method. The actual step of decorating the client proxy might take the following form:

```
18 EchoClient mixin add ::template::ProxyDecorator
```

Once this is achieved, calls to the accessor "glueobject" of the equally named attribute won't deliver the original glue object, but the decorator's one. This is due to the basic mixin mechanism provided by XOTcl with the decorator's "glueobject" method shadowing the actual

accessor called "glueobject". After the successful decoration, the call, again, is delivered to the endpoint stipulated in the second glue object.

We want to remind the reader at this point that the above classification somehow need to be considered oversimplifying. The main reason is the distinctions and elaborations are based on constructs that were defined, as an explicit design decision, as public interface (see Section 4.2). However, as many features are built upon generic XOTcl mechanisms, the same statements and considerations are equally valid for other entities or interface levels: Any object of type ::xotcl::Object may be associated with a glue object and can hold proxy methods through ::xotcl::Object→glue/::xotcl::Object→ad_glue, each of them being able to reference a distinct glue object. Therefore, the account given above applies to more than the scenarios outlined in the scope of this introductory section.

Besides, the current implementation comes with a couple of minor that might not be obvious to the developer. In particular, we do not distinguish between per-object or per-instance scopes. That is, a glue object specified on a class object will be valid for both its instances' and its own scope.

5 Internals

5.1 What are "service contracts"?

Service contracts, as implemented by the acs-service-contract package of the OpenACS core, introduces call abstraction, a concept with as many dimensions as actual areas of usage, to OpenACS as framework. The idea is not to reproduce existing documentation on service contracts, but to generalise the problem tackled by contracts. Service contracts aim at providing some sort of re-usability and a generic extension mechanism at the framework level. Call abstraction is a generic concept that can be found at the level of programming languages (of various flavours, OO and non-OO, functional etc.) and as infrastructure facilities in frameworks of various kind. The general motivation for call abstraction is a twofold: First, we want to assemble complex code blocks (i.e. a proc, involving sequences of calls to other code blocks) at design time (i.e. the time we conceptualise and write our program). Second, however, we do not want to specify a concrete addressee of some calls in our code block at run time. Call abstraction might be referred to as identity transparency which simply means the actual addressees or callees are not determined at design time! The simple motivation for these two requirements is that we want to allow varying behaviour within a more generally designed picture. Varying behaviour should be able to be introduced in a pre-defined and standardised manner. Let's take an example, taken from the OO-verse of XOTcl which gives a nice show case example, easier graspable than a general description of service contracts as such.

```
1 # / / / / / / / / / / / / / / /
2 # t0: introduce generic,
3 # extensible framework feature
4 ::xotcl::Object Indexer
5 Indexer proc index {} {
6   foreach instance [Indexable allinstances] {
7     puts [$instance getContent]
```

```
8       # dump the 'content' into the full-text index
9     }
10  }
```

Imagine, you set out to provide an full-text search in your application. In OpenACS, you might think of the "search" package as a direct example. An important role in this infrastructure module is played by an indexer that either on-demand or at regular intervals triggers the indexation of new text items. The indexer is represented by an XOTcl object "Indexer" that offers a method "index" responsible for the actual indexing walk-through. The fundamental design problem now is how to provide for the possibility to add new kinds of indexable items at an arbitrary point in time after the design and implementation of the indexer as such. To make it short, how to provide extensibility to the indexer? Key to an appropriate solution is the definition of a generic interface which needs to implemented by all indexable items that might be added in the future.

```
11  Class Indexable
12  Indexable abstract instproc getContent {}
```

This "caller interface", represented by the XOTcl class "Indexable" shown above, stipulates an abstract call "getContent" that needs to be resolved to a concrete call at runtime. Resolving means, in an OO setting, that implementing sub-classes of Indexable come with a non-abstract method "getContent". At t_0, i.e. design and implementation time of the infrastructure module, the per-object method "index" therefore exclusively refers to an abstract call "getContent". This is also known as template method, a prominent design pattern documented by [4].

```
13  Class ForumEntry -superclass Indexable
14  ForumEntry instproc render {} {
15    # return markup'ed content
16    return [self]([my info class])
17  }
18
19  ForumEntry instproc getContent {} {
20    return [my render]
21  }
```

As can be seen from the above snippet, by implementing "Indexable" and providing a concrete method "getContent" forum entries will be indexed upon future runs of the indexer, i.e. calls to "index". Registration, in our example, is realised by establishing a sub class relationship to Indexable. If, at t_1, wiki pages should be also be included in the full-text index, another specialised sub class of Indexable is needed:

```
22  Class XowikiPage -superclass Indexable
23  XowikiPage instproc render {} {
24    # return markup'ed content
25    return [self]([my info class])
26  }
27  XowikiPage instproc getContent {} {
28    return [my render]
29  }
```

This is a simple sketch of what extensibility is in this reading. Note, call abstractions in various settings come with different connotations (i.e. semantics), but they share basic ideas. Coming back to "service contracts" that realise call abstractions at a non-OO framework level, you might think of "service contracts" as abstract classes such as Indexable in the above example and of "service implementations" as registrars, represented by creating a sub class of Indexable. A more generic picture of call abstraction is given by Figure 7: The XOTcl

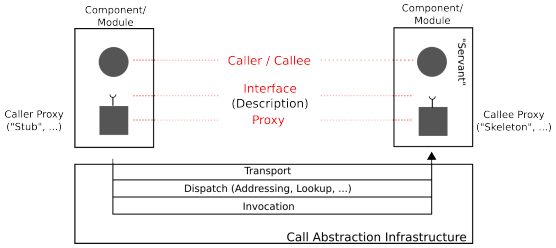

Figure 7: Call abstraction

example (and OpenACS service contracts) would translate into the following roles shown by the schematic outline: The class Indexable (a service contract such as FtsContentProvider) stipulates takes the role of interface (descriptions). In this role, they have two responsibilites, first interfacing to the caller (caller interface) and interfacing to the callee (callee interface). The caller is an arbitrary code block that issues the index call on objects of type Indexable. The callee, or servant, is the concrete getContent method of any sub class derived from class Indexable (e.g. XowikiPage, or, the service implementation content_revision). The proxy role is taken by the getContent method defined on Indexable and declared abstract (in OACS service contracts, these are operations defined on contracts).

A final clarification is needed at this point: What has become known as "remote procedure calls" (RPC) or "remote method invocations" (RMI) is the basic idea of call abstraction, but identity transparency is accompanied by location transparency.

6 Feature sets

6.1 xorb

6.2 xosoap

7 Roadmap

7.1 xorb

7.2 xosoap

8 Appendix

8.1 Examples - xosoap

Listing 1: A simple SOAP client

```
# -) prerequisites
namespace import ::xosoap::client::*
namespace import ::xorb::stub::*

# 1-) create and populate a 'glue' object
set glueObject [SoapGlueObject new \
                    -endpoint "http://websrv.cs.fsu.edu/~engelen/interop2.cgi"\
                    -action "http://websrv.cs.fsu.edu/~engelen/interop2.cgi" \
                    -callNamespace http://soapinterop.org/ \
                    -messageStyle ::xosoap::RpcEncoded]

# 2-) create a 'client proxy'
ProxyObject EchoClient -glueobject $glueObject

# 3-) create the interface of the 'client proxy'
EchoClient ad_proc -returns xsFloat \
    echoFloat {-inputFloat:xsFloat} {
      By calling this proc, a remote call is issued \
          against the previously defined endpoint
    } {}

# 4-) issue the call
ns_write [EchoClient echoFloat -inputFloat 1.6180339887]
```

Listing 2: An even simpler SOAP client

```
# -) prerequisites
namespace import ::xosoap::client::*

# 1-) create a combined 'client proxy' + 'glue' object
SoapObject EchoClient\
    -endpoint "http://websrv.cs.fsu.edu/~engelen/interop2.cgi"\
    -action "http://websrv.cs.fsu.edu/~engelen/interop2.cgi" \
    -callNamespace http://soapinterop.org/ \
    -messageStyle ::xosoap::RpcEncoded

# 2-) create the interface of the 'client proxy'
EchoClient ad_proc -returns xsFloat\
    echoFloat {
      -inputFloat:xsFloat
    } {
      By calling this proc, a remote call is issued \
          against the previously defined endpoint
    } {}

# 3-) issue the call
ns_write [EchoClient echoFloat -inputFloat 1.61805]
```

Listing 3: An XOTcl Object as SOAP client

```
# -) prerequisites
namespace import ::xorb::stub::*
namespace import ::xosoap::client::*

# 1-) create a 'glue' object
set glueObject [SoapGlueObject new \
                    -endpoint "http://websrv.cs.fsu.edu/~engelen/interop2.cgi" \
                    -action "http://websrv.cs.fsu.edu/~engelen/interop2.cgi" \
                    -callNamespace http://soapinterop.org/ \
                    -messageStyle ::xosoap::RpcEncoded]

# 2-) create an ordinary XOTcl object and turn it into a proxy client
::xotcl::Object EchoClient -glueobject $glueObject

# 3-) create the interface of the 'client proxy' using ad_glue keyword
# Note the structure of the declarative call, it does not take a method body!
EchoClient ad_glue -returns xsFloat\
    proc echoFloat {
      -inputFloat:xsFloat,glue
    } {
      By calling this proc, a remote call is issued \
          against the previously defined endpoint
    }

# 3-) issue the call
ns_write [EchoClient echoFloat -inputFloat 1.61805]
```

Listing 4: Using proxy templates/ template methods (I)

```
# -) prerequisites
namespace import ::xosoap::client::*
namespace import ::xorb::stub::*

# 1-) create and populate a 'glue' object
set glueObject [SoapGlueObject new \
                    -endpoint "http://websrv.cs.fsu.edu/~engelen/interop2.cgi"\
                    -action "http://websrv.cs.fsu.edu/~engelen/interop2.cgi" \
                    -callNamespace http://soapinterop.org/ \
                    -messageStyle ::xosoap::RpcEncoded]

# 2-) create a 'client proxy'
ProxyObject EchoClient -glueobject $glueObject

# 3-) create the interface of the 'client proxy'
EchoClient ad_proc -returns xsFloat \
    echoFloat {-inputFloat:xsFloat,glue} \
    {
      By calling this proc, a remote call is issued \
          against the previously defined endpoint. \
          However, this time, the call only gets executed \
          when the floating number matches a certain value \
          space.
    } {
      if {[expr {round(0.5 * (1 + sqrt(5)))}] == [expr round($inputFloat)]} {
```

```
        next;# invoke on remote method / procedure
      }
    }

# 4-) issue the call
ns_write [EchoClient echoFloat -inputFloat 1.6180339887]
```

Listing 5: Using proxy templates/ template methods (II)

```
# -) prerequisites
namespace import ::xosoap::client::*
namespace import ::xorb::stub::*

# 1-) create and populate a 'glue' object
set glueObject [SoapGlueObject new \
                    -endpoint "http://websrv.cs.fsu.edu/~engelen/interop2.cgi"\
                    -action "http://websrv.cs.fsu.edu/~engelen/interop2.cgi" \
                    -callNamespace http://soapinterop.org/ \
                    -messageStyle ::xosoap::RpcEncoded]

# 2-) create a 'client proxy'
ProxyObject EchoClient -glueobject $glueObject

# 3-) create the interface of the 'client proxy'
EchoClient ad_proc -returns xsFloat \
    echoFloat {
      -inputFloat:xsFloat,glue
      -nonpositionalArgument
       positionalArgument1
       positionalArgument2
   } \
    {
      By calling this proc, a remote call is issued \
          against the previously defined endpoint. \
          However, this time, the call only gets executed \
          when the floating number matches a certain value \
          space.
    } {
      ns_write "Variables in this scope: [info vars]\n"
      if {[expr {round(0.5 * (1 + sqrt(5)))}] == [expr round($inputFloat)]} {
        next;# invoke on remote method / procedure
      }
    }

# 4-) issue the call
ns_write [EchoClient echoFloat \
             -nonpositionalArgument "npValue" \
             -inputFloat 1.6180339887 \
             "val1" "val2"]
```

```
# / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / /
# $Id: example-07-soap-provider-init.tcl 73 2007-08-26 14:01:55Z ssoberni $

# -) prerequisites
namespace eval ::xorb::manual {
```

```
  namespace import ::xorb::*

  # 1-) provide an interface description: 'service contract'
  ServiceContract EchoService -defines {
    Abstract echoFloat \
        -arguments {
          inputDate:xsFloat
        } -returns returnValue:xsFloat \
        -description {
          Here, we outline an abstract call "echoFloat"
          and its basic characteristics to be realised
          both by servant code and client proxies.
        }
  } -ad_doc {
    This contract describes the interface of the
    EchoService example service as introduced by
    xorb's manual.
  }

  # 1a-) deploy your interface description
  EchoService deploy

  # 2-) Provide 'servant' code and register it with the invoker:
  # 'service implementation'
  ServiceImplementation EchoServiceImpl \
      -implements EchoService \
      -using {
        Method echoFloat {
          -inputFloat:required
        } {Echoes an incoming float} {
          return $inputFloat
        }
      }

  # 3a-) deploy your service implementation
  EchoServiceImpl deploy
}
```

Listing 6: Per-method precedence

```
# -) prerequisites
namespace import ::xosoap::client::*

# 1-) we define a distinct glue object

set glueObject [SoapGlueObject new \
                    -endpoint "http://dietrich.ganx4.com/nusoap/testbed/
                         round2_base_server.php"\
                    -messageStyle ::xosoap::RpcEncoded]

# 2-) create a combined 'client proxy' + 'glue' object
SoapObject EchoClient\
    -endpoint "http://websrv.cs.fsu.edu/~engelen/interop2.cgi"\
    -action "http://websrv.cs.fsu.edu/~engelen/interop2.cgi" \
    -callNamespace http://soapinterop.org/ \
    -messageStyle ::xosoap::RpcEncoded
```

```
# 3-) create the interface of the 'client proxy'
EchoClient ad_proc -glueobject $glueObject -returns xsFloat\
    echoFloat {
      -inputFloat:xsFloat
    } {
      By calling this proc, a remote call is issued \
          against the previously defined endpoint
    } {}

# 4-) issue the call
ns_write [EchoClient echoFloat -inputFloat 1.61805]
```

Listing 7: Shared scope

```
# -) prerequisites
namespace import ::xosoap::client::*
namespace import ::xorb::stub::*

# 1-) create and populate a 'glue' object
set glueObject [SoapGlueObject new \
                    -endpoint "http://websrv.cs.fsu.edu/~engelen/interop2.cgi"\
                    -action "http://websrv.cs.fsu.edu/~engelen/interop2.cgi" \
                    -callNamespace http://soapinterop.org/ \
                    -messageStyle ::xosoap::RpcEncoded]

# 2-) create a 'client proxy'
ProxyObject EchoClient -glueobject $glueObject

# 3-) create the interface of the 'client proxy'
EchoClient ad_proc -returns xsFloat \
    echoFloat {-inputFloat:xsFloat} {
      By calling this proc, a remote call is issued \
          against the previously defined endpoint
    } {}

# 4-) issue the call
ns_write [EchoClient echoFloat -inputFloat 1.6180339887]\n

# 5-) specify a decorator (a mixin class in XOTcl terms)

::xotcl::Class ProxyDecorator
SoapGlueObject ProxyDecorator::GlueObject \
    -endpoint "http://dietrich.ganx4.com/nusoap/testbed/round2_base_server.php"\
    -messageStyle ::xosoap::RpcEncoded
ProxyDecorator instproc glueobject args {
  return [self class]::GlueObject
}

# 6-) decorate the client proxy

EchoClient mixin add ::template::ProxyDecorator

# 7-) re-issue the call

ns_write [EchoClient echoFloat -inputFloat 1.6180339887]
```

```
# / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / /
# $Id: example-10-soap-provider-init.tcl 73 2007-08-26 14:01:55Z ssoberni $

# -) prerequisites

namespace eval :some_name_space {
  proc servantProc {inputFloat} {
    return $inputFloat
  }
}

namespace eval ::xorb::manual {
  namespace import ::xorb::*

  # 1-) provide an interface description: 'service contract'
  ServiceContract EchoService -defines {
    Abstract echoFloat \
        -arguments {
          inputDate:xsFloat
        } -returns returnValue:xsFloat \
        -description {
          Here, we outline an abstract call "echoFloat"
          and its basic characteristics to be realised
          both by servant code and client proxies.
        }
  } -ad_doc {
    This contract describes the interface of the
    EchoService example service as introduced by
    xorb's manual.
  }

  # 1a-) deploy your interface description
  EchoService deploy



  # 2-) Provide 'servant' code and register it with the invoker:
  # 'service implementation'
  ServiceImplementation EchoServiceImpl \
      -implements EchoService \
      -using {
        Delegate echoFloat \
            -per-object true \
            -private_p true \
            -for ::some_name_space::servantProc \
            {Echoes an incoming float}
      }

  # 3a-) deploy your service implementation
  EchoServiceImpl deploy
}
```

References

[1] Zolera SOAP Infrastructure (ZSI). URL http://pywebsvcs.sourceforge.net/.

[2] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes Second Edition. W3C Recommendation, W3C, 2004. URL http://www.w3.org/TR/xmlschema-2/.

[3] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thattle, and Dave Winer. Simpe Object Access Protocol (SOAP) 1.1. W3C Note, W3C, 2000. URL http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns – Elements of Reusable Object-Oriented Software. Addison Wesley Professional Computing Series. Addison Wesley, October 1994.

[5] Allan Kelly. Encapsulated Execution Context. In Proceedings of EuroPLoP 2003, Workshop D, 2003. URL http://hillside.net/europlop/europlop2003/papers/WorkshopD/D2_KellyA.rtf.

[6] Kai-Uwe Mätzel and Walter Bischofberger. The Any Framework. In Proceedings of the 2nd USENIX Conference on Object-Oriented Technologies and Systems, Toronto, Canada, June 1996. 179 – 190.

[7] Peter Sommerlad and Marcel Rüedi. Do-it-yourself Reflection. Presented to the Third European Conference on Pattern Languages of Programming and Computing (EuroPLoP'98), Bad Irsee, Germany, 1998. URL http://hillside.net/europe/EuroPatterns/files/DIY_Reflection.pdf.

[8] Markus Völter, Michael Kircher, and Uwe Zdun. Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware. Software Design Patterns. John Wiley & Sons Ltd., Chichester, England, 2005.

[9] Uwe Zdun. Patterns of argument passing. In Proceedings of the 4th Nordic Conference of Pattern Language of Programs (VikingPLoP2005), pages 1 – 25, Otaniemi, Finland, 2005. URL http://www.infosys.tuwien.ac.at/Staff/zdun/publications/arguments.pdf.