A Gentle Introduction to XOTcl SOAP

Stefan Sobernig

February 14, 2008

Overview

Introduction

What is XOTcl SOAP aka xosoap?

- SOAP consumer and provider infrastructure, currently in release version 0.4.3 (February 2008).
- SOAP 1.1 compliance (SOAP 1.2 is work-in-progress); SOAP marshaler / demarshaler on top of tdom.
- Auto-generation of WSDL 1.1 and, optionally, WS-I compliant interface descriptions
- Support for various WSDL 1.1 marshaling styles: Rpc/Encoded, Rpc/Literal. Document/Literal is work-in progress.
- Support for XML Schema primitive and composite types based on an extensible type infrastructure.
- Framework interoperability: Designed to be compliant to SOAPBuilder Interoperability Lab test suites, currently A + B.

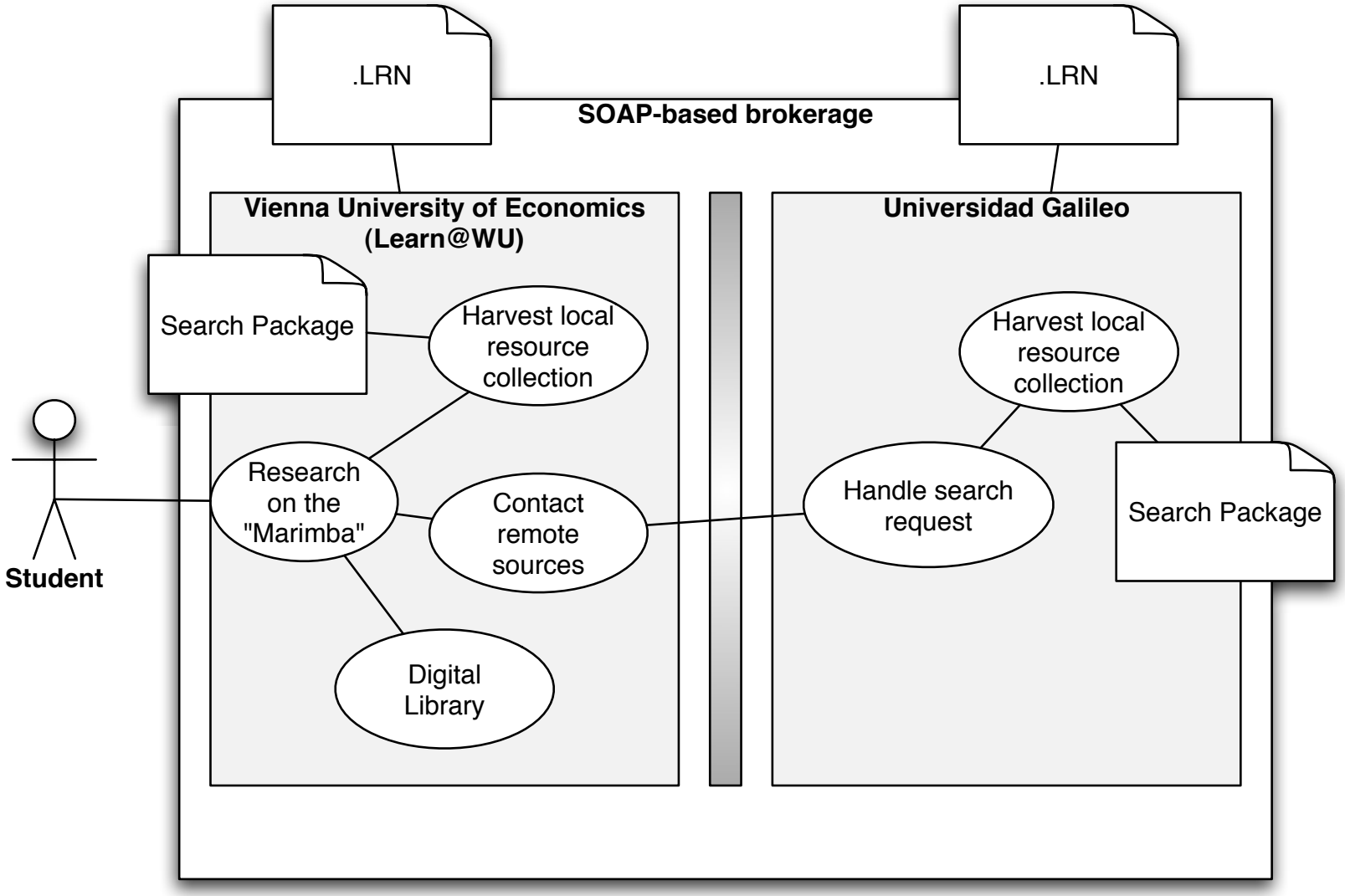
What is the XOTcl Request Broker aka xorb?

- Generic brokerage Völter et al. [2005] infrastructure for OpenACS, based on XOTcl and xotcl-core.
- Allows for plugging-in protocol extensions: currently SOAP support by xosoap.
- Based upon an object-oriented layer and extension to OpenACS service contracts, allowing for a more agile use of contracts and implementations
- Allows for publishing existing Tcl and XOTcl code as remoting, e.g. SOAP, services.
- Support for legacy code through ”interface adapters”
- Generic extension mechanism through ”interceptors”
- Fine-grain facilities for invocation access control
- Tight integration with XOTcl idioms

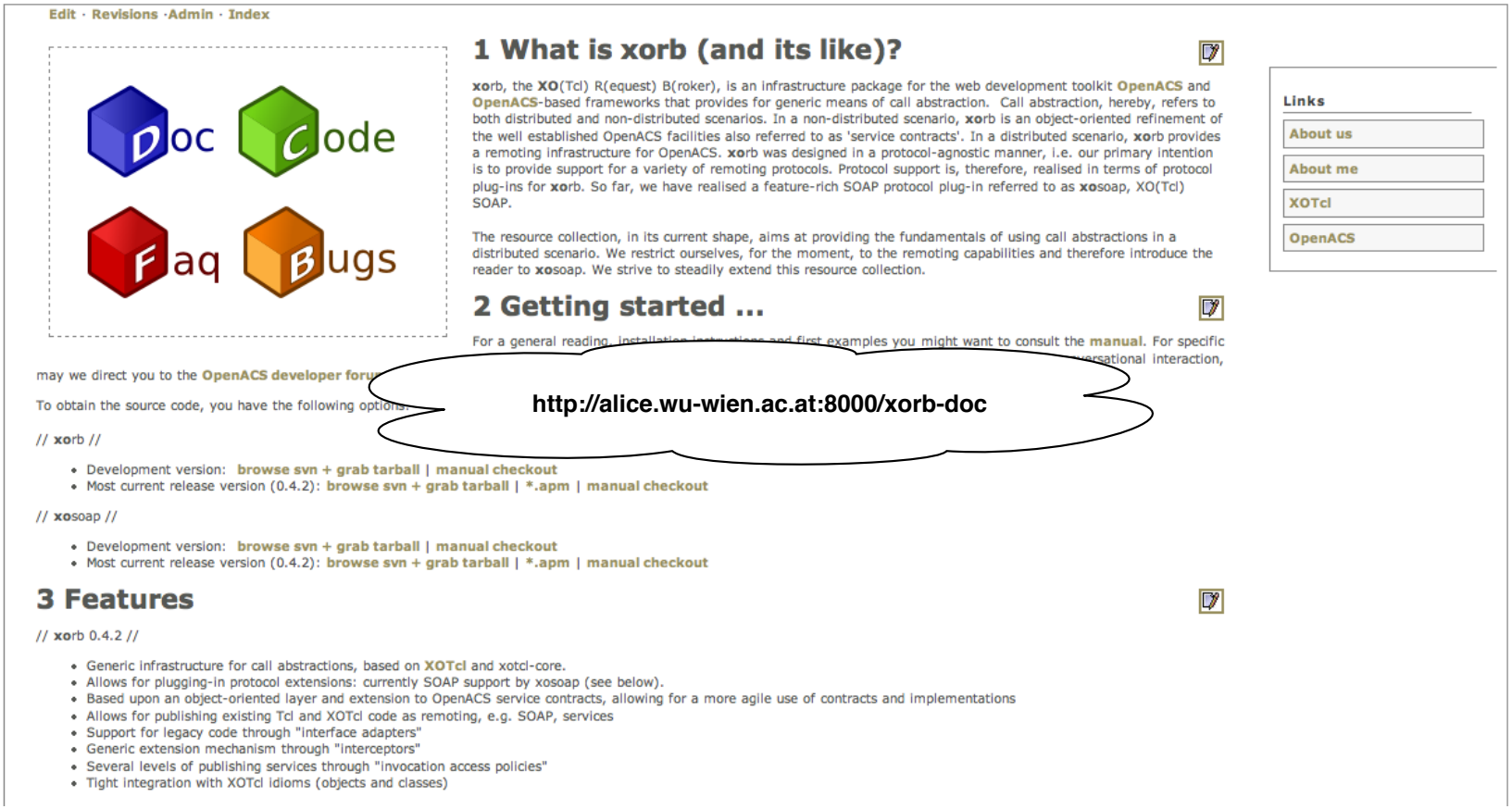
Profile of this tutorial

- The tutorial is built around a demo OpenACS application package: xosoap-demo
- Following a simple use case story, realised in the package, we are going introduce you to our broker Völter et al. [2005] infrastructure and their interfaces.
- Many drivers to this efforts: Applied ones from research projects, more theory-driven motivation from my thesis project.
- In the scope of this tutorial, I won't touch the generic framework, rather how to use the SOAP protocol plugin (xotcl-soap) available.
- The objective to outline the fundamental steps to get you started using our infrastructure packages with minimum effort. The skeleton package is at your disposal (see slide on Resources).

Our demo story

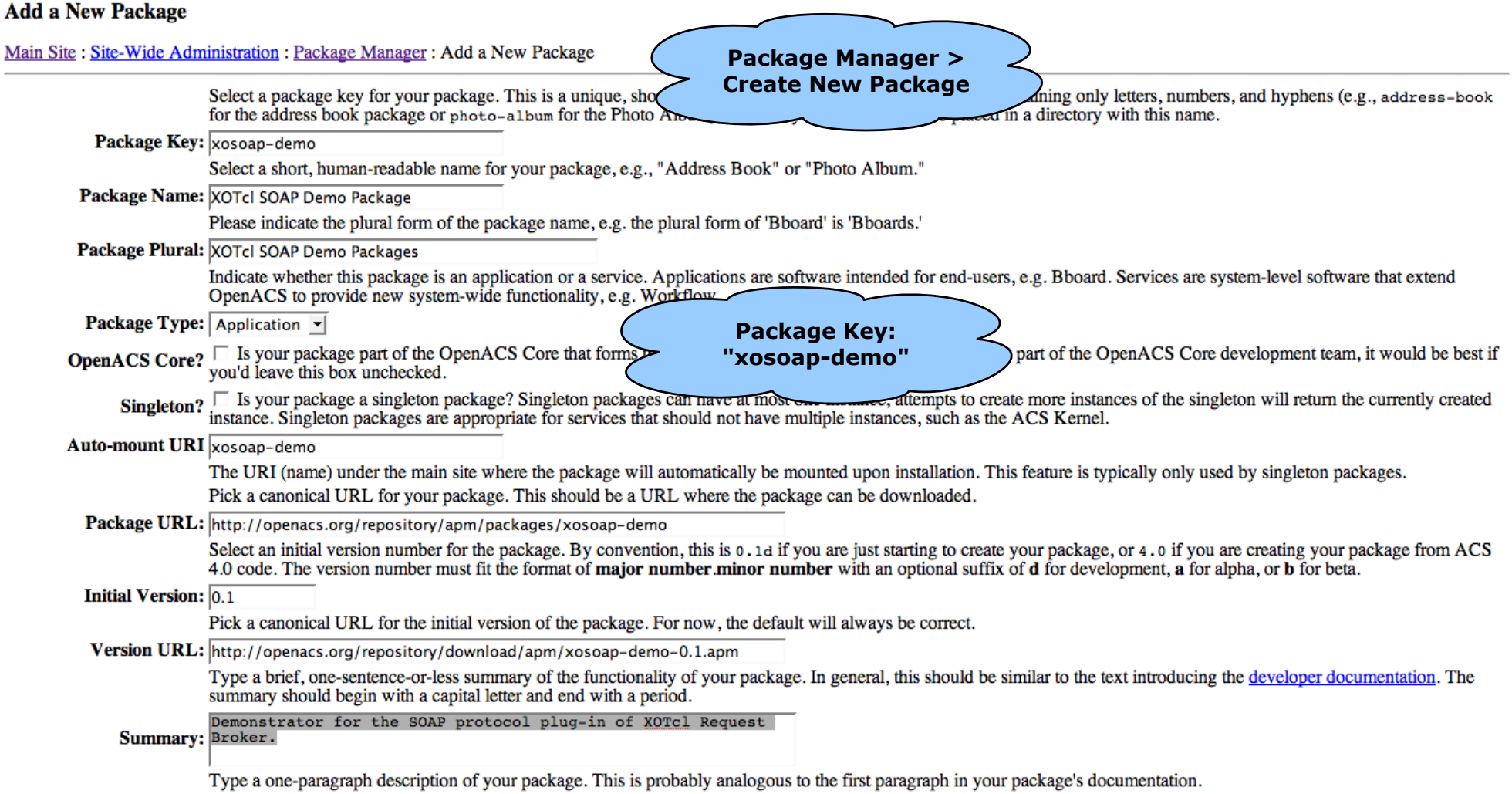


Resources needed

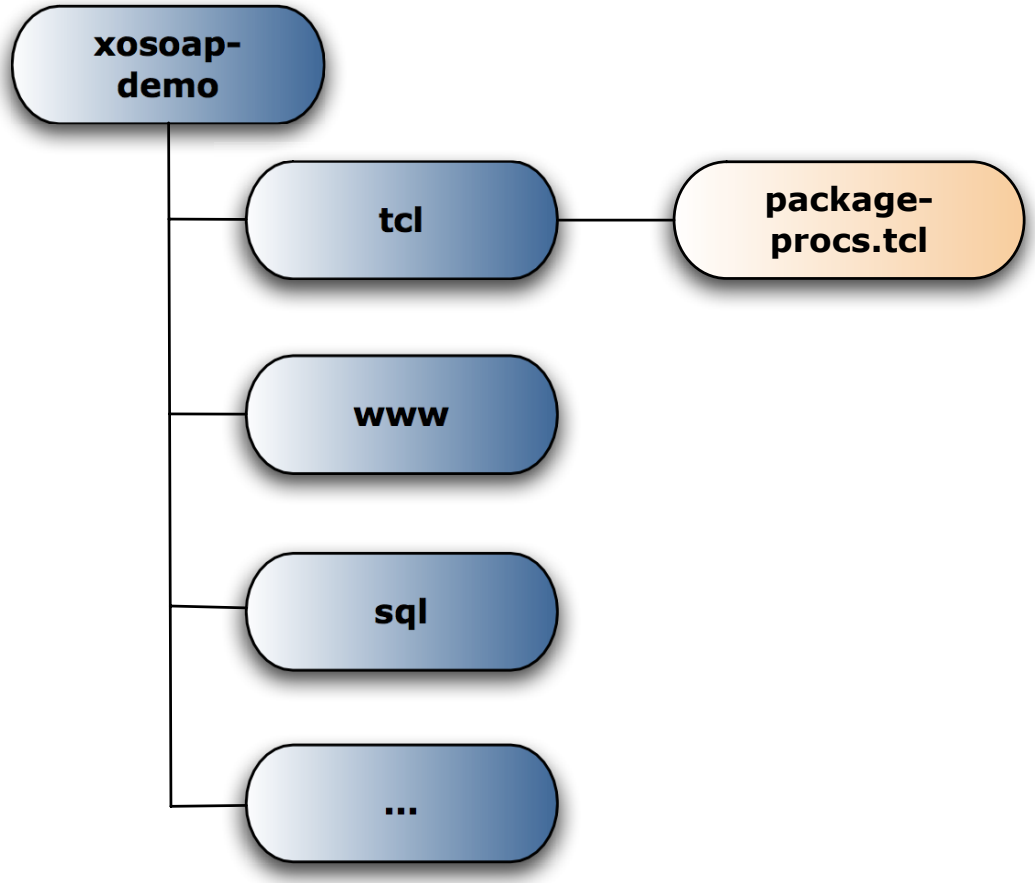


Your first xosoap-enabled package

Prerequisites / Create package structure (1)



Prerequisites / Create package structure (2)



Prerequisites / Create package manager (1)

```
# / / / / / / / / / / / / / / / / / / / / / / / / / /
# "::xo::library" is a powerful alternative
# to "ad_library". Most importantly, it allows
# to specify dependencies between library scripts
# to circumvent the default lexicographic order
# of evaluation.

::xo::library doc {

    Package infrastructure for the xosoap demonstrator
    package.

    @creation-date 2008-02-14
    @author Stefan Sobernig
    @cvs-id $Id$
}
```

Prerequisites / Create package manager (2)

```
# / / / / / / / / / / / / / / / /
# Declaring an application-specific
# namespace is good practice ...

namespace eval ::demo {

  # ...

}
```

Prerequisites / Create package manager (3)

```
# / / / / / / / / / / / / / / / / / / / / / / / / / / / /
# We define a package manager for our demo package.
# Package managers are provided by the XOTcl Core
# and act as convenient helpers when dealing with
# OpenACS APM-style packages.
# Our package manager class may be addressed as
# "::demo::Package".

::xo::PackageMgr create Package \
    -superclass ::xo::Package \
    -pretty_name "XOTcl SOAP Demo Package" \
    -package_key "xosoap-demo"

# / / / / / / / / / / / / / / / / / / / / / / / / / / / /
# We provide a per-instance constructor which
# may be used to specify initialisation behaviour
# for instances of our Package Manager.

Package instproc init {} {
    # initialisation magic
}
```

Common pitfalls

Is it important to consider the order of package initialisation when naming my new packge, i.e. picking a *package key*?

- NO, XOTcl Core and xorb provide means to explicitly require package dependencies, regardless of the lexicographic initialisation order.

Is it mandatory to use XOTcl Core package management, e.g. a package manager, for my package?

- NO, but it facilitates your development task if the complexity starts to increase.

Is it mandatory to provide for a Tcl namespace for my package?

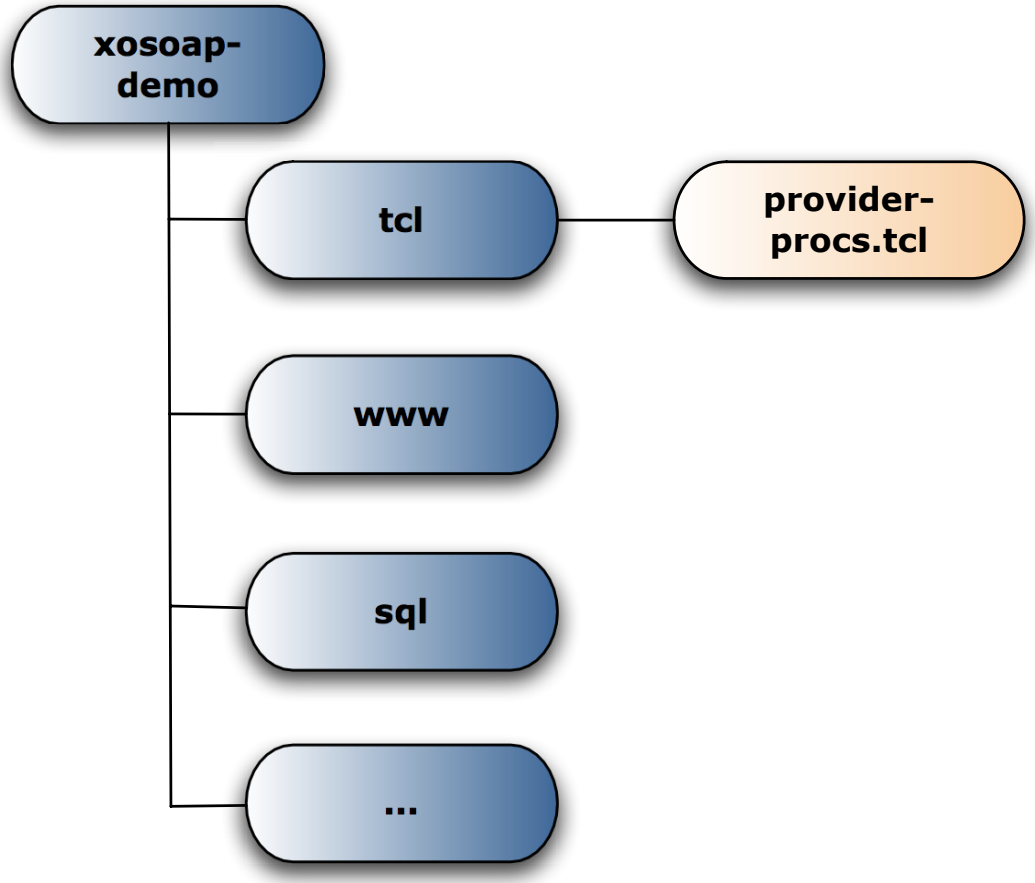
- Not necessarily, as the containing Tcl namespace does not convey any critical semantics (from the perspective of XOTcl Core or xorb/xosoap) at this point. However, it may be considered good practice. In the context of defining your SOAP provider, the choice of namespace becomes an issue (see below).

Your first SOAP provider

Provider / Our recipe

- To begin with, we provided for a library and package environment for our provider code to be hosted with (see above).
- Then, we look at devising an explicit interface which stipulates the public behaviour of our SOAP provider.
- Once defined, we look at realising the interface as a provider-side specification object (“service contract”).
- In addition, we have to provide a reference implementation for the materialised interface, a so-called “service implementation”. It acts either as servant or provider-side proxy for a servant.
- Finally, we look at some the requirement of explicit deployment.

Provider / Create a provider library script (1)



Provider / Create a provider library script (2)

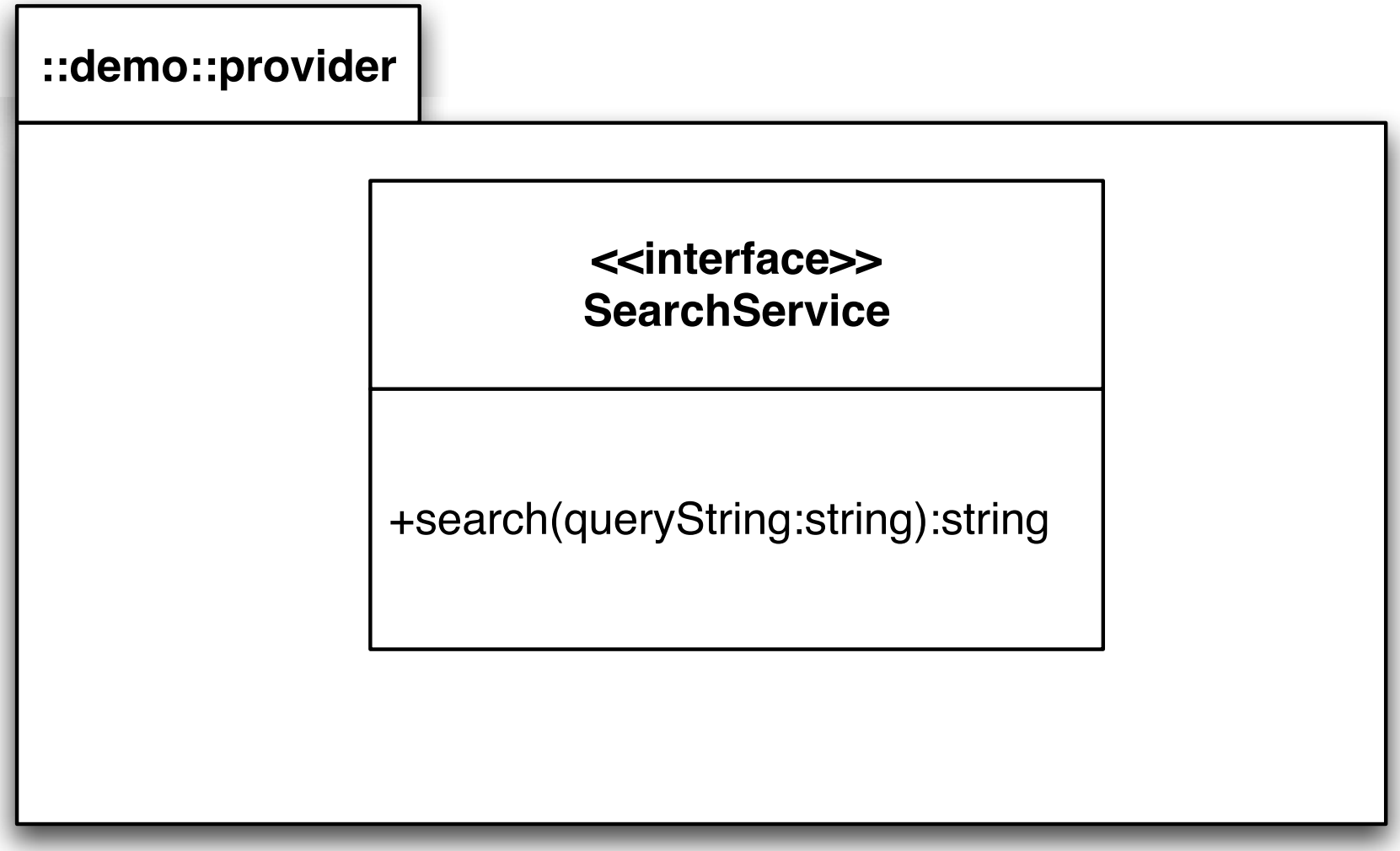
```
# / / / / / / / / / / / / / / / / / / / / / / / / / /
# "::xo::library" is a powerful alternative
# to "ad_library". Most importantly, it allows
# to specify dependencies between library scripts
# to circumvent the default lexicographic order
# of evaluation.

::xo::library doc {

    Library script hosting our SOAP provider

    @creation-date 2008-02-14
    @author Stefan Sobernig
    @cvs-id $Id$
}
```

Provider / Provide xorb in the package scope

```
# / / / / / / / / / / / / / / / / / / / / / / / / / / / /
# Upon server initialisation, OpenACS packages are
# sourced in their lexicographic order. Core packages
# (e.g. acs-*, xotcl-core), however, have priority
# and are processed before non-core ones. This allows
# packages as ours to draw upon their code even at
# initialisation time. Now, remember, our package
# is named "xosoap-demo" which ranks before xorb
# ("xotcl-request-broker") and xosoap ("xotcl-soap").
# Therefore, we have to explicitly require xorb
# before declaring our SOAP provider by using
# "::xo::db::require package <package_key>".
# Having explicitly required xorb, you may use facilities
# residing in the "::xorb::*" namespace.

::xo::db::require package xotcl-request-broker
```

Provider / Sketch an explicit interface

A conceptual sketch of the *explicit interface* embodied by our SOAP provider:



Provider / Providing for dedicated namespace

```
# / / / / / / / / / / / / / / / /
# Declaring an provider-specific
# namespace is >recommended< practice

namespace eval ::demo::provider {

namespace import ::xorb::*

# SOAP provider specification goes here ...

}
```

Provider / Realise the Interface as Service Contract

```
# / / / / / / / / / / / / / / / / / / / / / / / / /
# 1st step: Provide a specification of an >explicit<
# interface. This may be achieved by instantiating
# "::xorb::SericeContract". This yields a special-purpose
# XOTcl class object that represents the specification
# for our demo SearchService.

ServiceContract SearchService -defines {
    ::xorb::Abstract search \
        -arguments {
            queryString:xsString
        } -returns "returnValue:xsString" \
        -description {
            A generic interface that provides
            a "search" operation to callers.
        }
}
```

Provider / Provide an Implementation

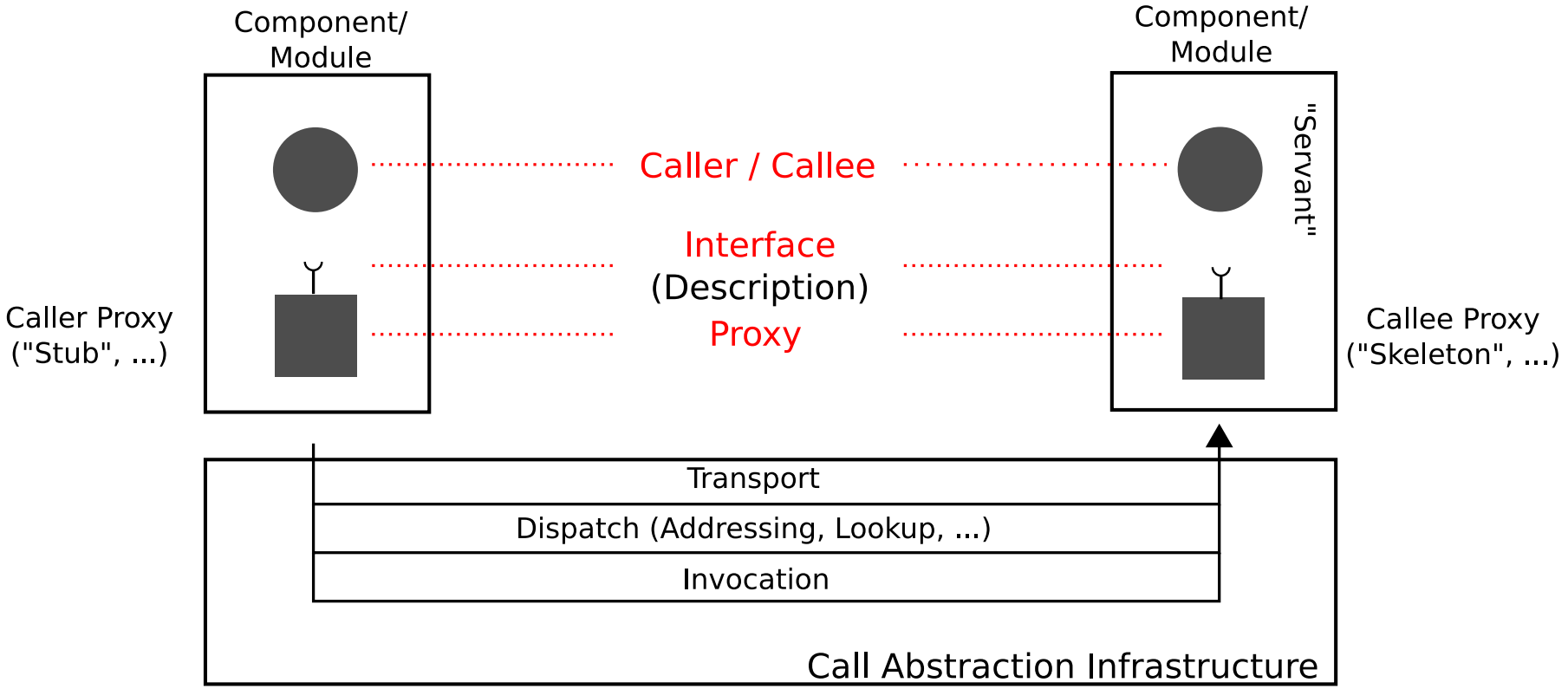
```
# / / / / / / / / / / / / / / / / / / / / / / / / / / / /
# 2nd step: Provide a sample implementation that
# realises (implements) the above >explicit interface<
# by re-using the OpenACS Search package infrastructure.
# For this task, we create an object of type
# "::xorb::ServiceImplementation".

ServiceImplementation OpenACSSearchPackageImpl \
    -implements SearchService \
    -using {
        # / / / / / / / / / / / / / /
        # Method: search
        Method search {
            -queryString:required
        } {
            This method takes the query string,
            and performs the actual search by calling
            the responsible Search package facility.
        } {
            set result "A rose is a rose is a rose"
            # / / / / / / / / / / / / / / / / / / / / / / / / /
            # Here, we would need to ressemble the
            # behaviour of either search/www/search.tcl
            # <or> revert to using tsearch2::search, for
            # instance, directly ...
            return $result
        }
    }
```

Provider / Deploy Interface & Implementation

```
# / / / / / / / / / / / / / / / / / / / / / / / / / /
# 3rd step: Finally, you need to >deploy< both the
# interface and its reference implementation by
# sending a deploy() message. Deployment involves
# a set of minor tasks, such as registering with the
# Invoker and checks for interface conformity.

SearchService deploy
OpenACSSearchPackageImpl deploy
```

Provider / Excursus / OpenACS Service Contracts (1)

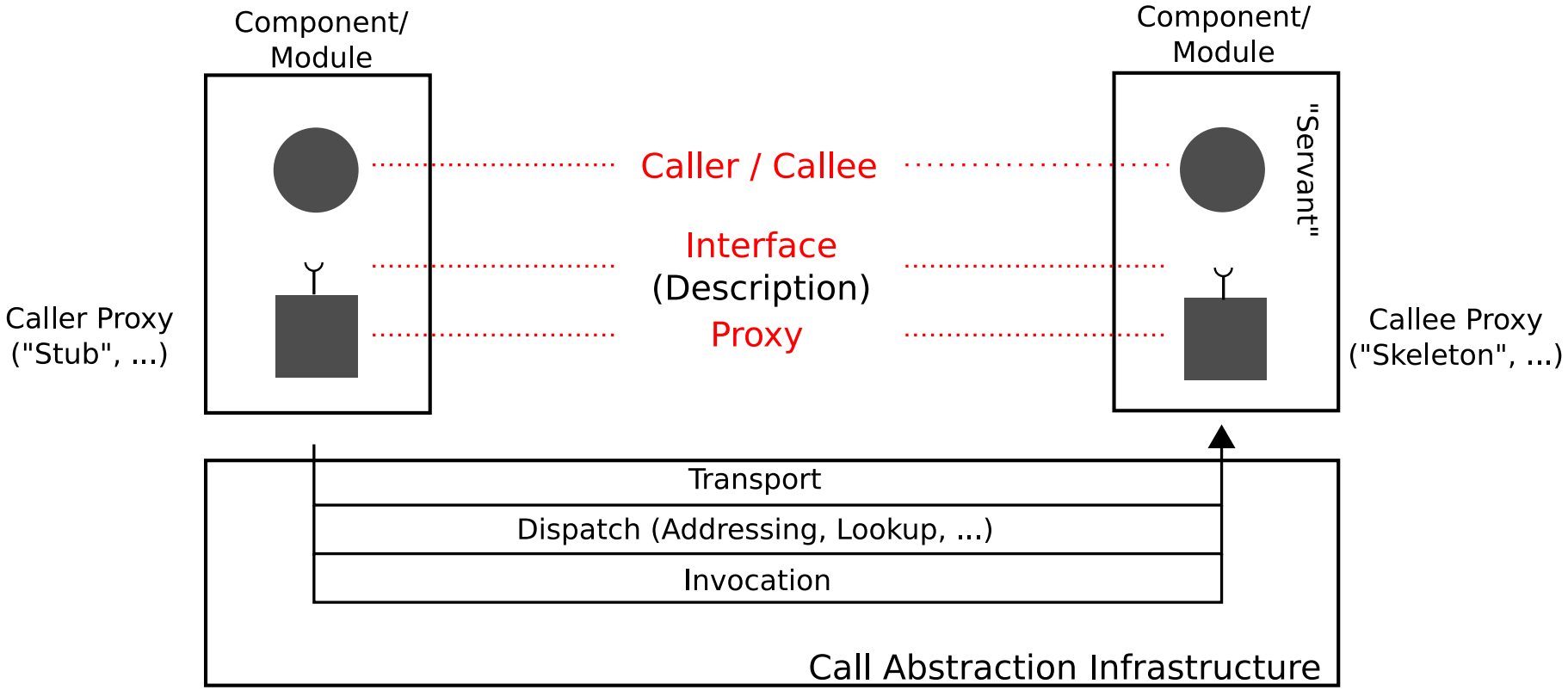
Service Contracts realise an *indirection layer* as framework extension strategy:



Conceptually, this has been labelled EXPLICIT INTERFACE Buschmann and Henney [2003], at a more implementation level BRIDGE pattern Gamma et al. [1994].

Provider / Excursus / OpenACS Service Contracts (2)

The Search Package is a primary example for the use of service contracts:



Provider / Excursus / Tcl names and broker references

A key task of BROKERS, as xorb, is bridging between *references* or *identifiers* across various scopes:

- The ultimate target reference (to the servant) managed by xorb, i.e. its INVOKER is :: demo::provider::OpenACSSearchPackageImpl
- xosoap transliterates these into an URI scheme, according to the following rules:
  - Default URI scheme: (site—node)/services/(tcl—qualifiers)/(object—name), e.g. /xosoap/services/demo/provider/OpenACSSearchPackageImpl
  - The (tcl—qualifiers) fragment takes care of the ambiguity between top-level ("global") xorb and legacy OpenACS contracts or implementations:
    - At the level of xorb and the service contracts, entity names "::myContract" and "myContract" (as allowed for legacy ones) are logically distinct.
    - The mapping into a URI would represent both by /xosoap/services/myContract which would be a fundamental conflict.
    - Therefore, there is a (configurable) default URI segment (default:"acs") for legacy items so we can address both. "myContract" becomes /xosoap/services/acs/myContract

Provider / Excursus / Deployment

... serves a couple of purposes

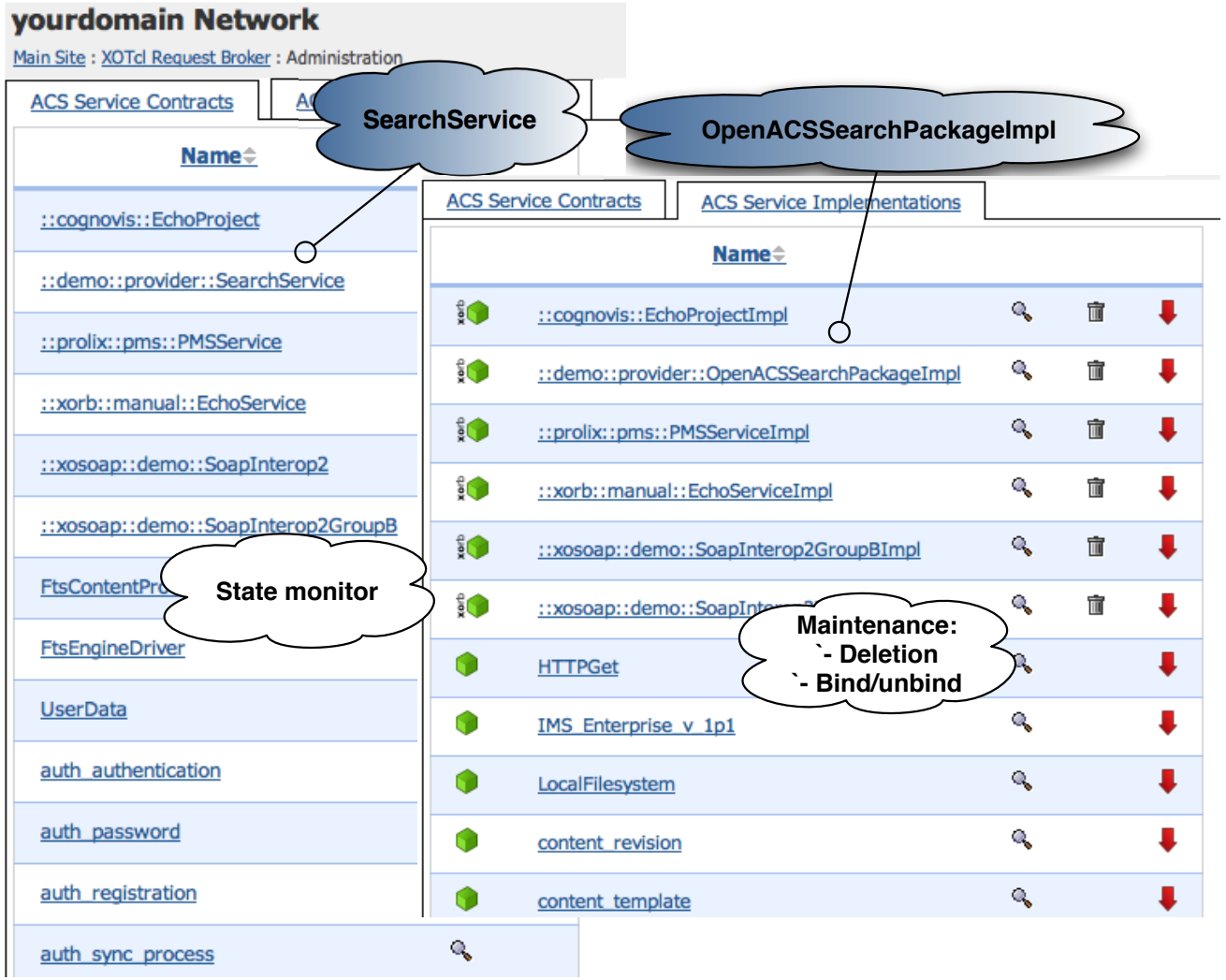
- Verify the consistency / correspondence of an implementation to the interface description (service contract). Currently, we enforce a limited type of *behavioural containment*.
- Introduces a stage life-cycle which differentiates between prototyping, accomplishing, and publishing a provider.
- The process of deleting either a service contract or service implementation is linked to the deployment call. As contracts/ implementations are persisted, one need to remove the deploy call and then clear the back-end from the persisted representations.

Provider / What to keep

- There are four steps involved:
  1. Provide for dedicated Tcl Namespace, e.g. :: demo::provider
  2. Create a ServiceContract class object, realising your interface sketch; e.g. :: demo::provider::SearchService
  3. Create a ServiceImplementation class object, realising your interface sketch; e.g. :: demo::provider::OpenACSSearchPackageImpl
  4. Deploy the latter two . . .
- There is some magic that turns Tcl qualified names in URIs for the scope of xosoap and vice versa . . .
- Conceptually, XOTcl Request Broker and its plug-ins build upon OpeACS core framework features (“service contracts”) and simply turn them inside-out!

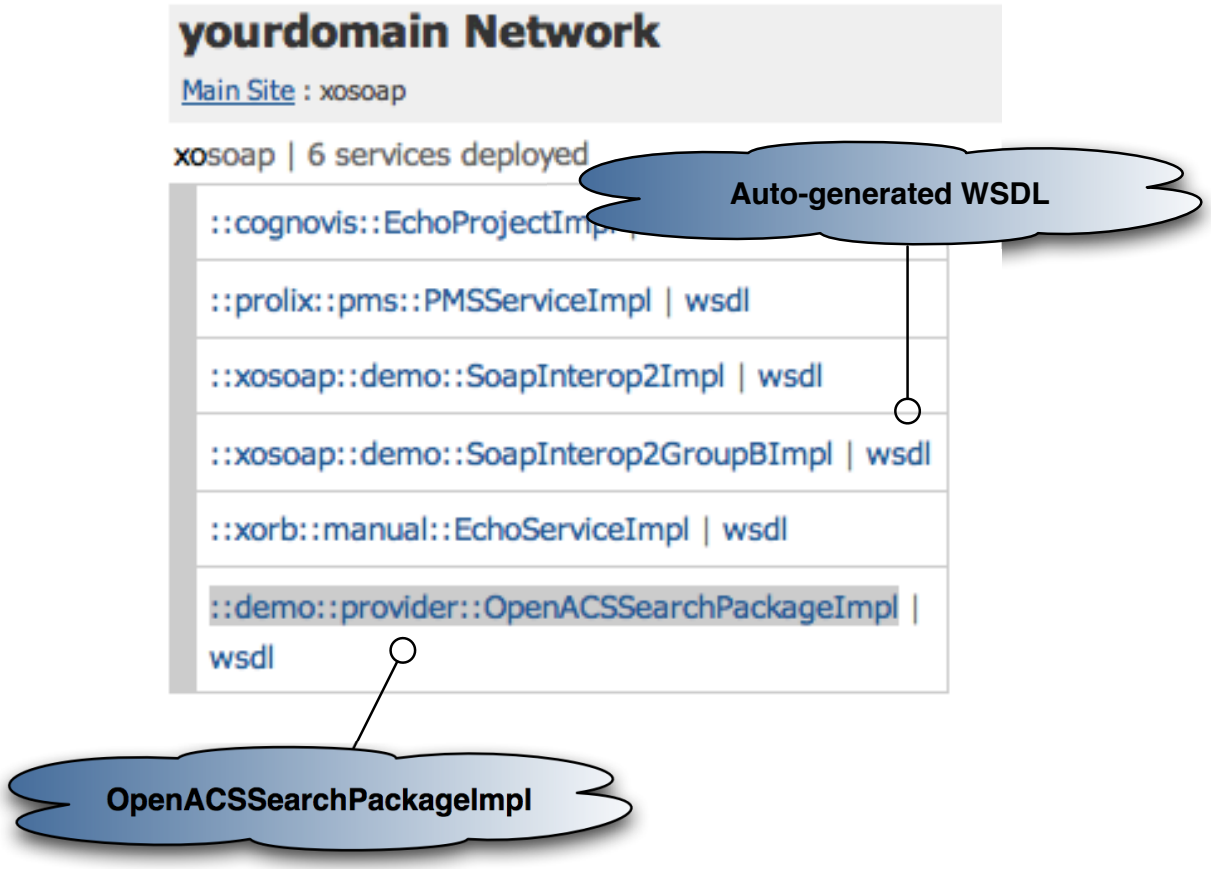
Provider / Maintenance (1)

The xorb cockpit at /request-broker/admin



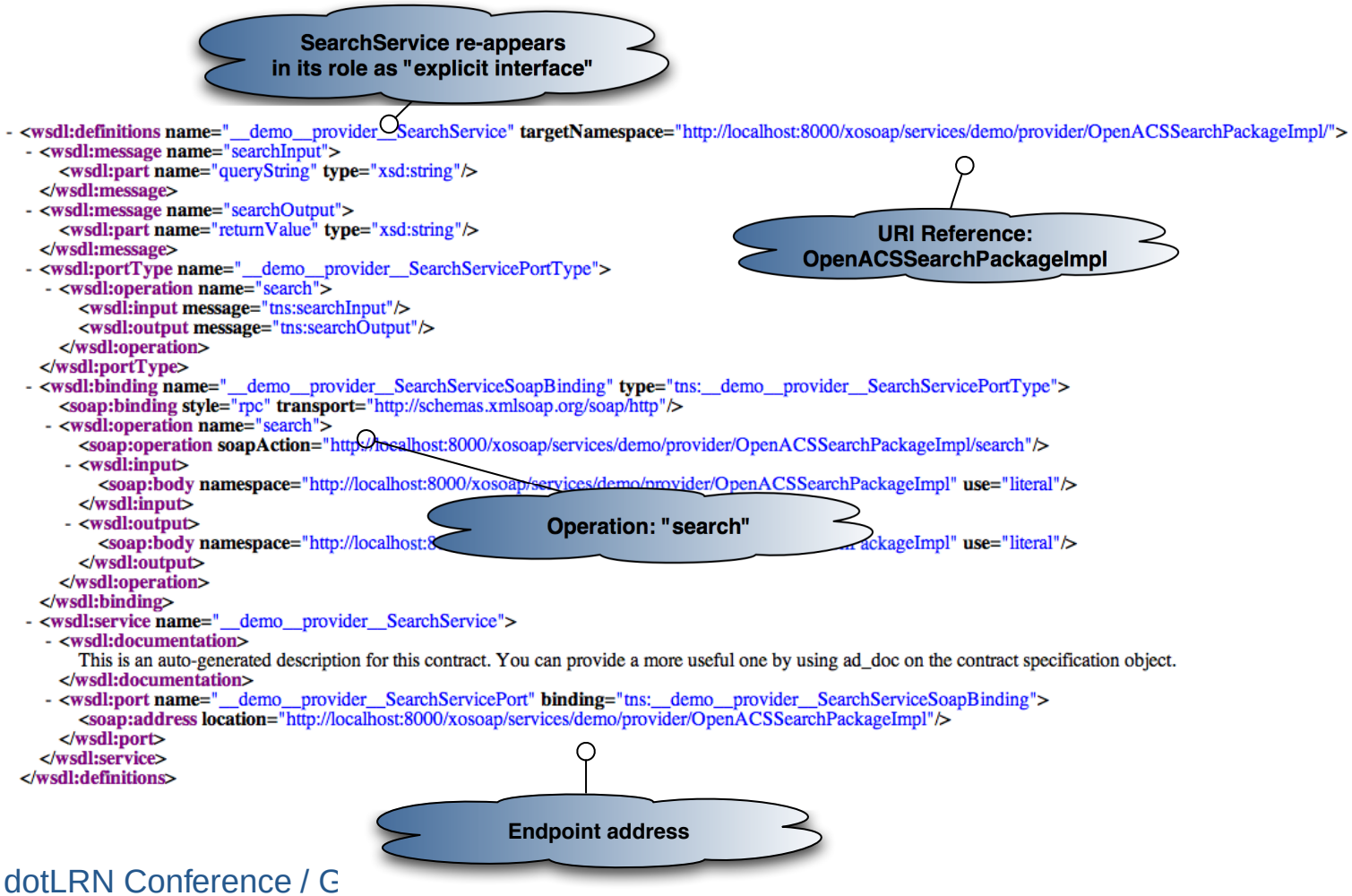
Provider / Maintenance (2)

The xosoap view of deployed implementations at /xosoap/services



Provider / Auto-generation of interface description (WSDL)

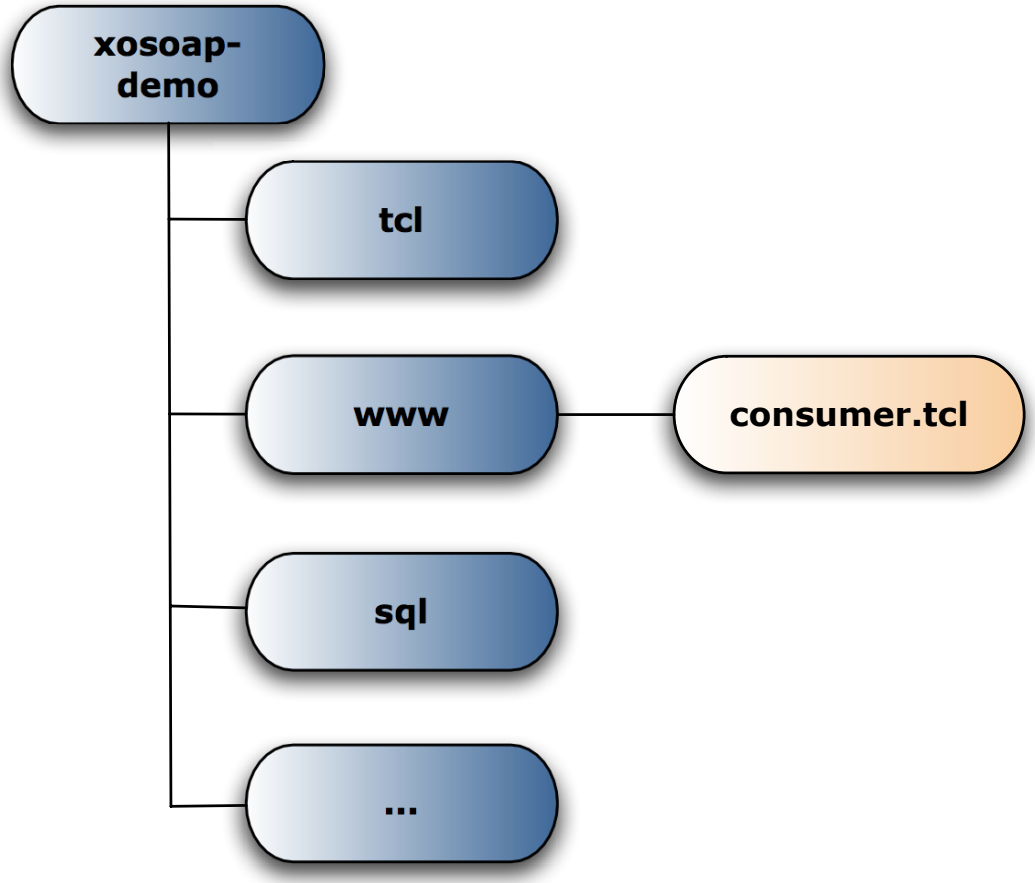
Based on your in-memory specification of an explicit interface, xosoap generates WSDL representations. Point your browser to e.g. /xosoap/services/demo/provider/OpenACSSearchPackageImpl?**s=wsdl** . . .




```
- <wsdl:definitions name="__demo__provider__SearchService" targetNamespace="http://localhost:8000/xosoap/services/demo/provider/OpenACSSearchPackageImpl/">
  - <wsdl:message name="searchInput">
      <wsdl:part name="queryString" type="xsd:string"/>
    </wsdl:message>
  - <wsdl:message name="searchOutput">
      <wsdl:part name="returnValue" type="xsd:string"/>
    </wsdl:message>
  - <wsdl:portType name="__demo__provider__SearchServicePortType">
    - <wsdl:operation name="search">
        <wsdl:input message="tns:searchInput"/>
        <wsdl:output message="tns:searchOutput"/>
      </wsdl:operation>
    </wsdl:portType>
  - <wsdl:binding name="__demo__provider__SearchServiceSoapBinding" type="tns:__demo__provider__SearchServicePortType">
      <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    - <wsdl:operation name="search">
        <soap:operation soapAction="http://localhost:8000/xosoap/services/demo/provider/OpenACSSearchPackageImpl/search"/>
      - <wsdl:input>
          <soap:body namespace="http://localhost:8000/xosoap/services/demo/provider/OpenACSSearchPackageImpl" use="literal"/>
        </wsdl:input>
      - <wsdl:output>
          <soap:body namespace="http://localhost:8
                                                    ackageImpl" use="literal"/>
        </wsdl:output>
      </wsdl:operation>
    </wsdl:binding>
  - <wsdl:service name="__demo__provider__SearchService">
    - <wsdl:documentation>
        This is an auto-generated description for this contract. You can provide a more useful one by using ad_doc on the contract specification object.
      </wsdl:documentation>
    - <wsdl:port name="__demo__provider__SearchServicePort" binding="tns:__demo__provider__SearchServiceSoapBinding">
        <soap:address location="http://localhost:8000/xosoap/services/demo/provider/OpenACSSearchPackageImpl"/>
      </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

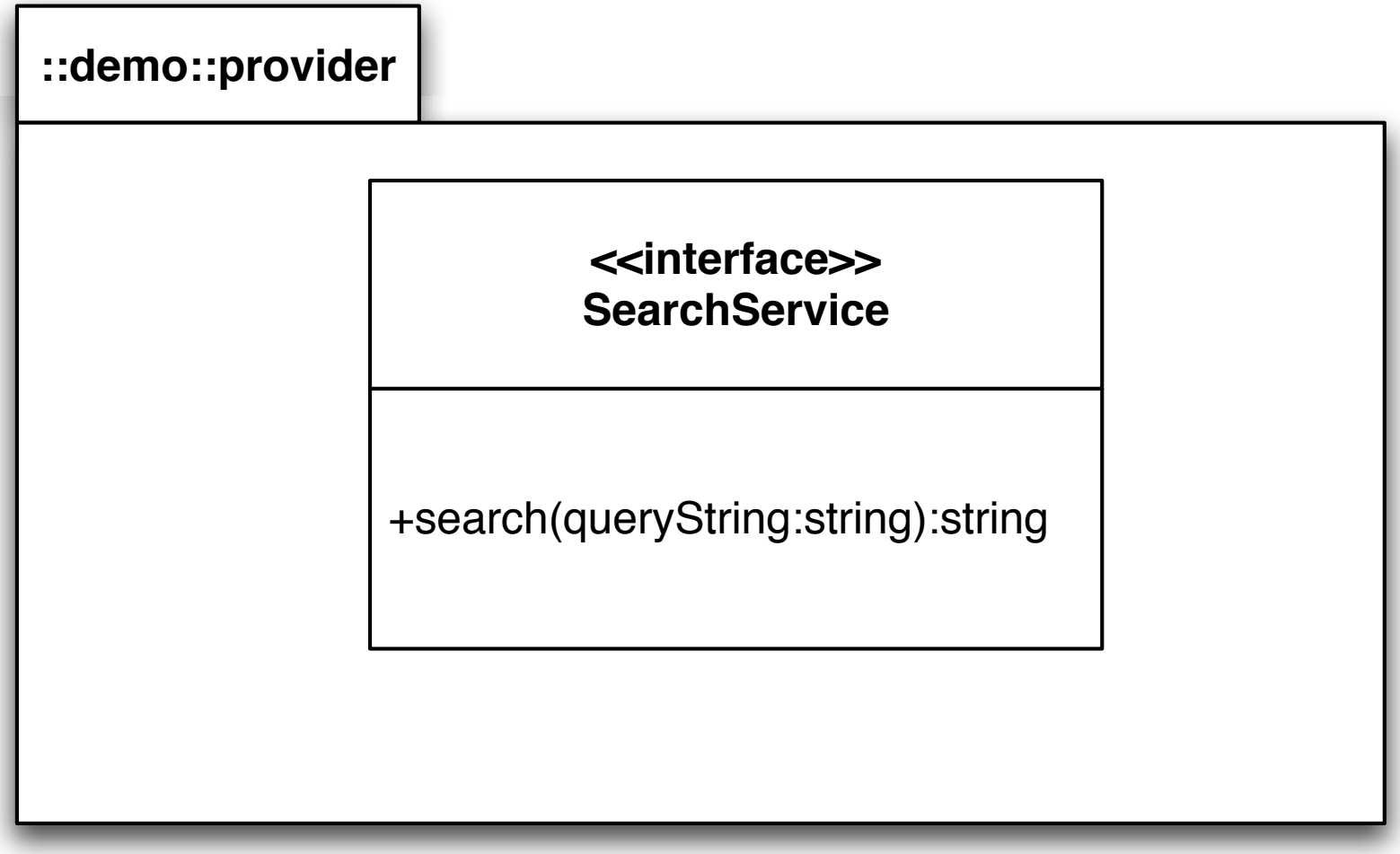
Your first SOAP consumer

Consumer / Create a “WUI” script



Consumer / Realise an explicit interface

While in the provider context, the explicit interface served as *callee interface*, it now "in-forms" potential callers, i.e. CLIENT PROXIES . . .



Consumer / Our recipe

- First, we provide a specific type of CONTEXT OBJECT that conveys two kinds of informations for our intended remoting interaction:
  1. Re-usable invocation information
  2. Invocation context information
- Second, the EXPLICIT INTERFACE, e.g. SearchService, needs to be realised for the consumer side.
- Parametrisation of call and performing actual call.

Consumer / Prerequisites

```
# / / / / / / / / / / / / / / / / / / / / / /
# Remember the package manager you created
# initially in this demo package! Now it
# is time to use it to contextualise requests
# to this sample script hosting a consumer.
# The call to initialize() resolves the current
# package context and allows for specifying
# parameter requirements on the debarking
# requests.

::demo::Package initialize -ad_doc {

    This is a sample W(eb) U(ser) I(nterface) script
    that demonstrates creating a basic SOAP consumer
    and handy XOTcl core features in this respect ...

    @date 2008-02-14
    @author Stefan Sobernig
    @cvs-id $Id$

} -parameter {
  {-queryString:required}
}
```

Consumer / A glue object

```
# / / / / / / / / / / / / / / / / / / / / / /
# First, we provide for a "glue" object that
# stores particular kind of invocation
# information, i.e. endpoint address, but
# is also carrier for invocation context
# information, required in more complex
# scenarios. Most importantly, the selection
# of the kind of glue object determines
# the remoting protocol used, ie.e SOAP.
# "SoapGlueObject" resides in the
# "::xosoap::client" namespace.

namespace import ::xosoap::client::*

set endpoint \
    http://localhost:8000/xosoap/services/demo/provider/OpenACSSearchPackageImpl

set glueObject [SoapGlueObject new \
                    -endpoint $endpoint\
                    -messageStyle ::xosoap::RpcLiteral]
```

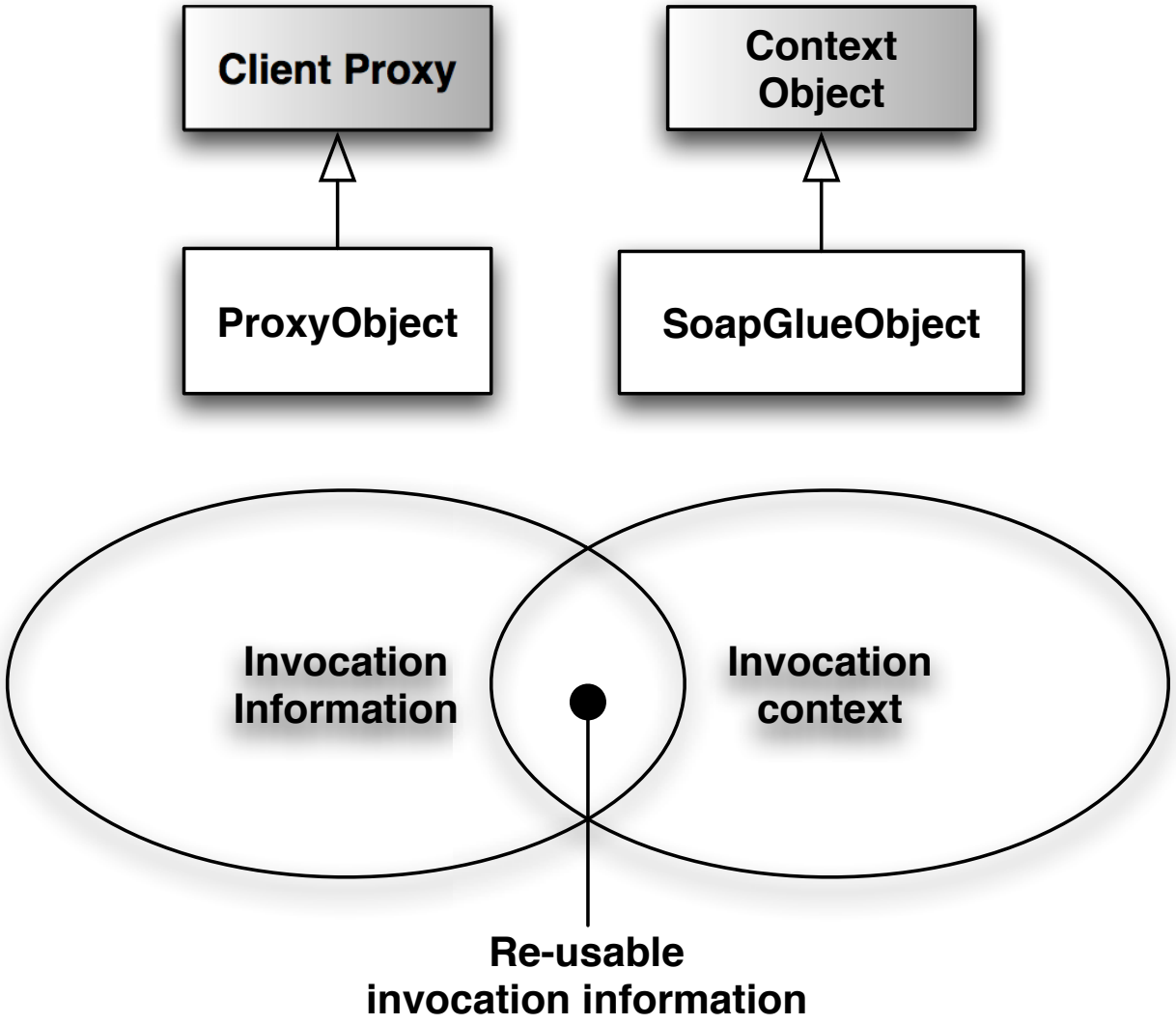
Consumer / a client proxy

```
# / / / / / / / / / / / / / / / / / / / / /
# Second, we realise the explicit interface
# by providing a counterpart to the
# ServiceImplementation ("skeleton") at the
# provider side, a "client proxy". Therefore,
# you need to import the habitants of
# "::xorb::stub::*" namespace into the current
# scope and create an object of type
# "ProxyObject". Note that the previously
# defined glueobject is passed by association
# to the client proxy!
namespace import ::xorb::stub::*

ProxyObject SearchServiceProxy -glueobject $glueObject

SearchServiceProxy ad_proc -returns xsString \
    search {-queryString:xsString} \
    {Implementation for the search operation} \
    {}
```

Consumer / The invocation

```
# / / / / / / / / / / / / / / / / / / / /
# Finally, we perform the invocation and
# assign the invocation result to a local
# Tcl variable that will populate the
# consumer.adp template ...
set html [SearchServiceProxy search \
              -queryString $queryString]
```

Consumer / Driving concepts



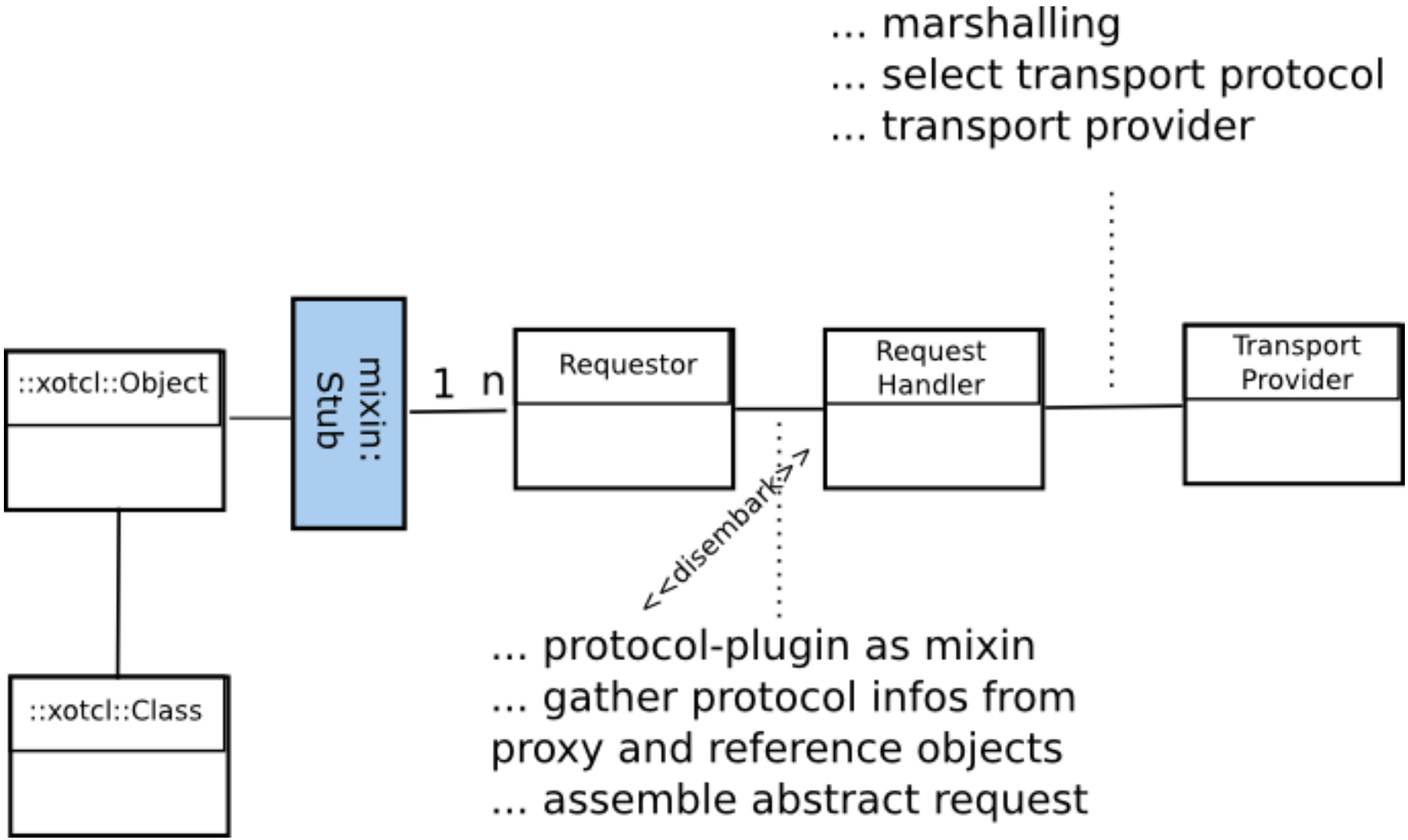
Consumer / Glue objects

- Encapsulate and organise request information needed at various layers and stages.
- It closely follows the idea of Context Objects as a strategy of argument passing.
- Using an object as a argument passing vehicle allows for:
  - handling of a huge variety of heterogeneous argument information needed to perform a call (protocol and transport layer).
  - transformation of argument information during handling (streaming)
  - a unspecified variety of clients to be served
- Glue objects are simply associated to objects, potentially turning them into client proxies.
- Glue objects are aligned to OO concepts: Glue objects can be linked to classes that provide them to their instances. Similarily, glue objects can be injected into existing object hierarchies (class tree) by means of mixins.

Consumer / Client proxy

- The role of client proxies is to mimic the interface of ’remotely’ listening/ hosted objects.
- Client proxies therefore represent the realisation of interface descriptions. They are responsible to resolve a ’glue’ object, translate their interface description into call information and pass the letter together with the glue object as actual invocation data.
- ’glue’ / ’ad_glue’ as keywords are the instruments of declaring such a proxy interface

Client proxies in context



Bibliography

References

- 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
- 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
- Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley & Sons Ltd., Chichester, England, 2000
- Our xorb/xosoap resource collection

Advanced Features

Overview

- Advanced indirection *invocation interceptors*
- Integrated exception and SOAP Fault handling
- Publishing legacy code: *adapters* available for Objects and Procedures.
- Rich variety of interfaces to use, ranging from close-to-XOTcl idioms to special-purpose citizens.

See the authoritative manual for details.