Chap 6. CORBA-based Architecture

Part 6.4 Implementing CORBA Applications

1. Introduction
2. CORBA Naming Service
3. Implementation: General Approach
4. Java Implementations
6. C++ Implementations
7. CORBA Trading Service

1. Introduction

-To enable distributed computing, CORBA specification addresses several key challenges such as *portability* and *interoperability*.

÷CORBA server and client applications may be started on different machines with exactly the same results.

÷Communication between the client and the server are handled transparently in a platform- and language-independent manner, provided there is a CORBA ORB available for the platform and programming languages involved.

-To achieve these functions, CORBA uses standard mechanisms, services, and protocols. Some of the most essential of these artifacts are the ORB and CORBA Services such as **Naming** and **Trading**.

÷The ORB acts as a mediator for any interaction between CORBA objects

÷The Naming Service and the Trading services are some of the key mechanisms used by the ORB to locate objects in the network.

-CORBA application programming relies on these standard mechanisms and the high-level abstraction of the object interfaces captured in IDL.

2. CORBA Naming Service

Motivation

•Object-oriented middleware uses object references to address server objects

-We need to find a way to get hold of these object references without assuming physical locations

•A name is a sequence of character strings that can be bound to an object reference

-A name binding can be resolved to obtain the object reference

•The CORBA ***Naming Service***:

- –Allows locating components by external names
- –Similar to white pages

•The CORBA ***Trading Service***:

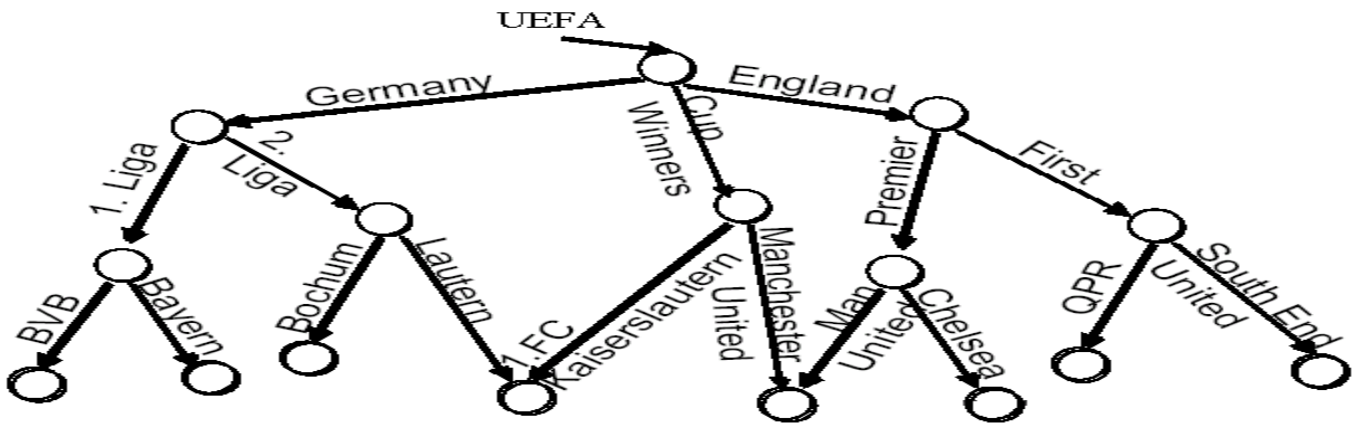
- –Locating components by service characteristics
- –Similar to yellow pages

Common Principles

- There may be many server objects in a distributed object system
- Server objects may have several names
  - Leads to large number of name bindings
- Name space has to be arranged in a hierarchy to avoid
  - Name clashes
  - Poor performance when binding/resolving names
- Hierarchy achieved by naming contexts

Naming Graph

- Names are composed of possibly more than one component
- Used to describe traversals across several naming contexts



-Example of name components: ("UEFA","England", "Premier", "Chelsea")

CORBA Naming Service

- Supports bindings of names to CORBA object references.

-Names are scoped in naming contexts.

-Multiple names can be defined for object references.

-Not all object references need names.

- Names are composed of simple names.

-Simple names are value-kind pairs.

-*Value* attribute is used for resolving names.

-*Kind* attribute is used to provide information about the role of the object.

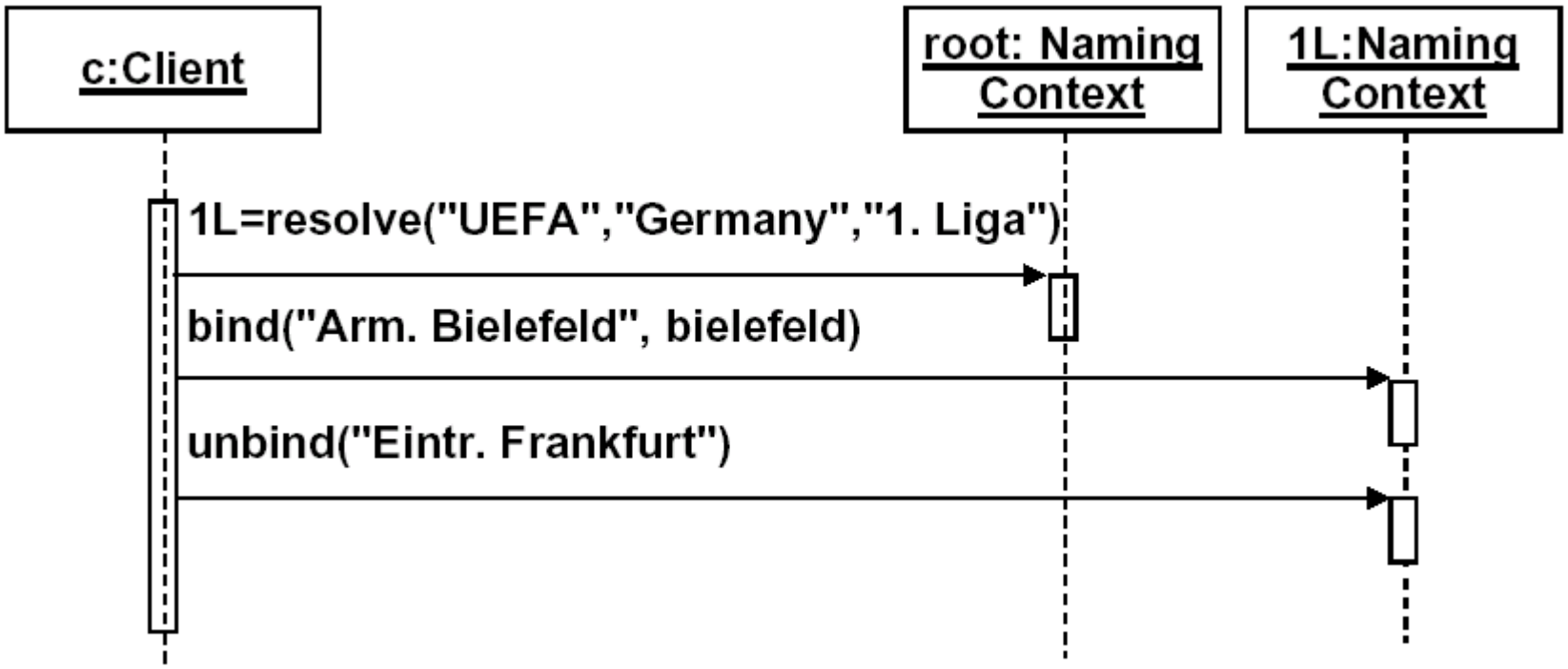
```
//IDL type for name
module CosNaming {
        typedef string Istring;
        struct NameComponent {
                Istring id;
                Istring kind;
        };
        typedef sequence <NameComponent> Name;
                ...
};
```

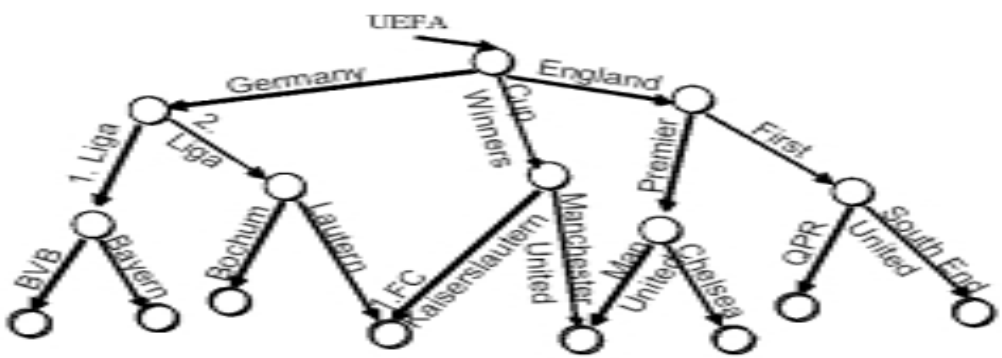
- Naming Service is specified by two IDL interfaces:
  - ***NamingContext*** defines operations to bind objects to names and resolve name bindings.
  - ***BindingIterator*** defines operations to iterate over a set of names defined in a naming context.

```
interface NamingContext {
    void bind(in Name n, in Object obj) raises (NotFound, ...);

    Object resolve(in Name n) raises (NotFound,CannotProceed,...);

    void unbind (in Name n) raises (NotFound, CannotProceed...);

    NamingContext new_context();
    NamingContext bind_new_context(in Name n) raises (NotFound, ...);

    void list(in unsigned long how_many,
                              out BindingList bl,
                              out BindingIterator bi);
    }
```

Example: Naming Scenario-Binding

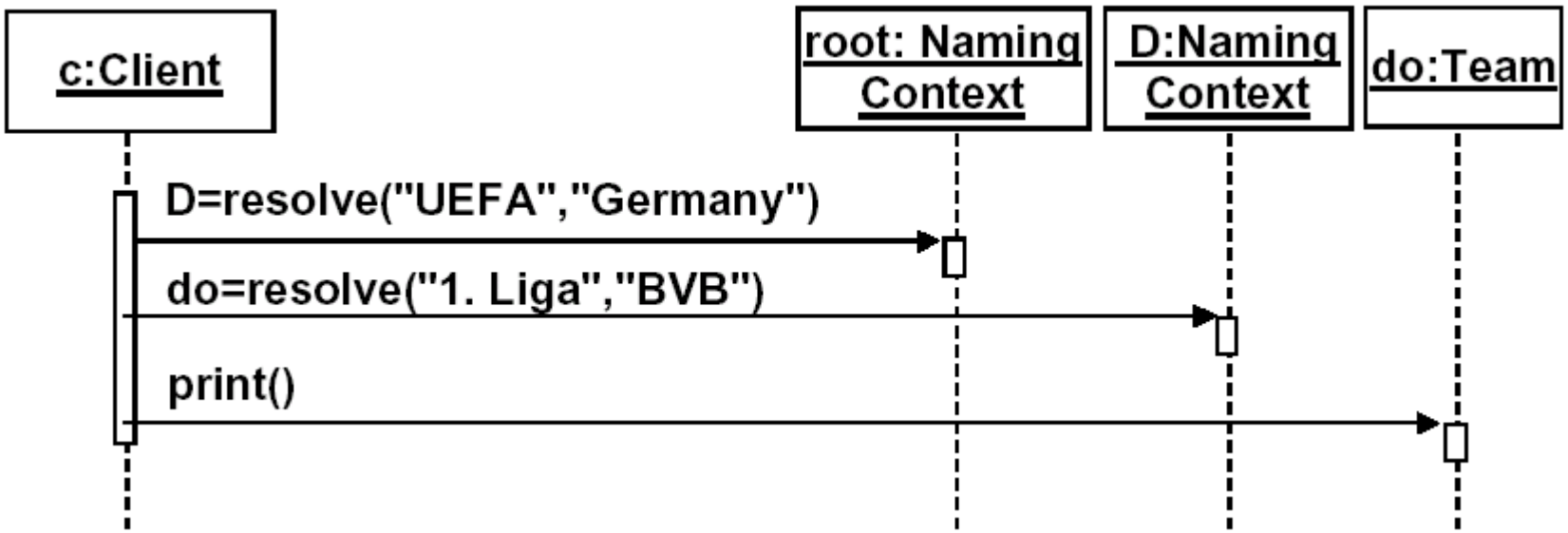
Promote Bielefeld to German '1. Liga' and relegate Frankfurt





Example: Naming Scenario-Resolving

Print squad of Borussia Dortmund



Bootstrapping Naming Service

-How to get the Root Naming Context?

÷ORB Interface provides for initialization

```
module CORBA {
        interface ORB {
                typedef string ObjectId;
                typedef sequence <ObjectId> ObjectIdList;
                exception InvalidName{};
                ObjectIdList list_initial_services();
                Object resolve_initial_references
                        (in ObjectId identifier)
                                raises(InvalidName);
        }
}
```

÷Initial object references are provided by the ORB via the **resolve_initial_references** operation.

÷The operation takes as argument the name of the service for which we need the initial reference.

÷The list of available service names may be returned by calling **list_initial_services** on the ORB.

÷ Standard service names include: ***NameService, TradingService***, *InterfaceRepository, SecurityCurrent, TransactionCurrent, DynAny Factory, ORBPolicyManager, PolicyCurrent,* ***RootPOA***, *POACurrent,* ***Component HomeFinder***.

Example: we want to register a servant instance with the name server.

Assume that we obtain a reference to the servant as:

Functions fref = …

We can proceed as follows:

//get the root naming context; use NamingContextExt instead of
//NamingContext-is part of the interoperable naming service (INS).

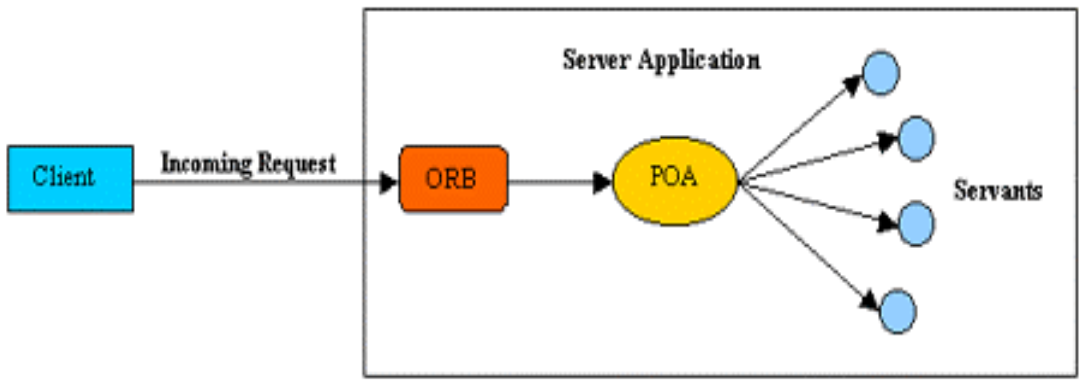
Object objRef= orb.resolve_initial_references("NameService");
NamingContextExt ncRef=NamingContextExtHelper.narrow(objRef);

//Create a name or the object and bind the Object Reference in Naming
String name ="Calc";
NameComponent path[] = ncRef.to_name(name);
ncRef.rebind(path,fref);

Portable Object Adapter (POA)

-Up to CORBA 2.1, the only standard object adapter defined by the OMG is the Basic Object Adapter (BOA), which provides basic services to allow a variety of CORBA objects to be created.

÷ORB vendors and developers, however, discovered that the BOA is ambiguous and missing some features. This led vendors develop their own proprietary extensions, which resulted in poor portability between different ORB implementations.



-The new standard object adapter is the Portable Object Adapter (POA), which provides new features that allow developers to construct object implementations that are portable between different ORB products supplied by different vendors.

÷The POA acts as a mediator between the ORB and the server application.

÷The client request to a target object is received by the ORB, which will dispatch the request to the POA that hosts the target object. The POA will then dispatch the request to the servant, which subsequently carries the request and sends the results back to the POA, to the ORB, and finally to the client.

÷Since an application may have multiple POAs, in order for the ORB to dispatch the request to the right POA, it uses an object key, which is an identifier that is part of the request that is kept in the object reference.

÷One part of the object key called the object ID is used by the POA to determine an association (such associations might be stored in a map) between the target object and a servant.

-The steps for using the POA may vary depending on the type of application being developed.

1. Get the Root POA

The first step is to get the root POA, which is managed by the ORB and provided to the application using the initial object name RootPOA, as follows:

```
ORB orb = ORB.init(argv, null);
POA rootPOA = POAHelper.narrow( orb.resolve_initial_references("RootPOA"));
```

2. Activate the POAManager

A POAManager is associated with one or more POA objects. It is responsible for controlling the processing state of the POAs. When a POAManager object is created, it is activated as follows:

```
rootPOA.the_POAManager().activate();
```

3. Create the Object Reference

Once an object reference is created in a server, it can be exported to clients. An object reference contains information related to object identity and others required by the ORB to identify and locate the server and the POA with which the object is associated.

```
FunctionsImpl f= new FunctionsImpl();
Functions fref = FunctionsHelper.narrow(rootpoa.servant_to_reference(f));
```

Example: Sample CORBA Server Code (Java)

//get reference to rootpoa & activate the POAManager

Object poaRef = orb.resolve_initial_references("RootPOA");

POA rootpoa = POAHelper.narrow(poaRef);

rootpoa.the_POAManager().activate();

//create servant (implementation object) and connect it with the ORB

FunctionsImpl f= new FunctionsImpl();

f.setORB(orb);

//get object reference from the servant

Object ref = rootpoa.servant_to_reference(f);

Functions fref = FunctionsHelper.narrow(ref);

```
//IDL code
module Calculator {
 interface Functions {
   float square_root (in float number);
   float power (in float base, in float exponent
   };
};
```

//get the root naming context; use NamingContextExt instead of //NamingContext-is part of the interoperable naming service (INS).

Object objRef= orb.resolve_initial_references("NameService");

NamingContextExt ncRef=NamingContextExtHelper.narrow(objRef);

//bind the Object Reference in Naming

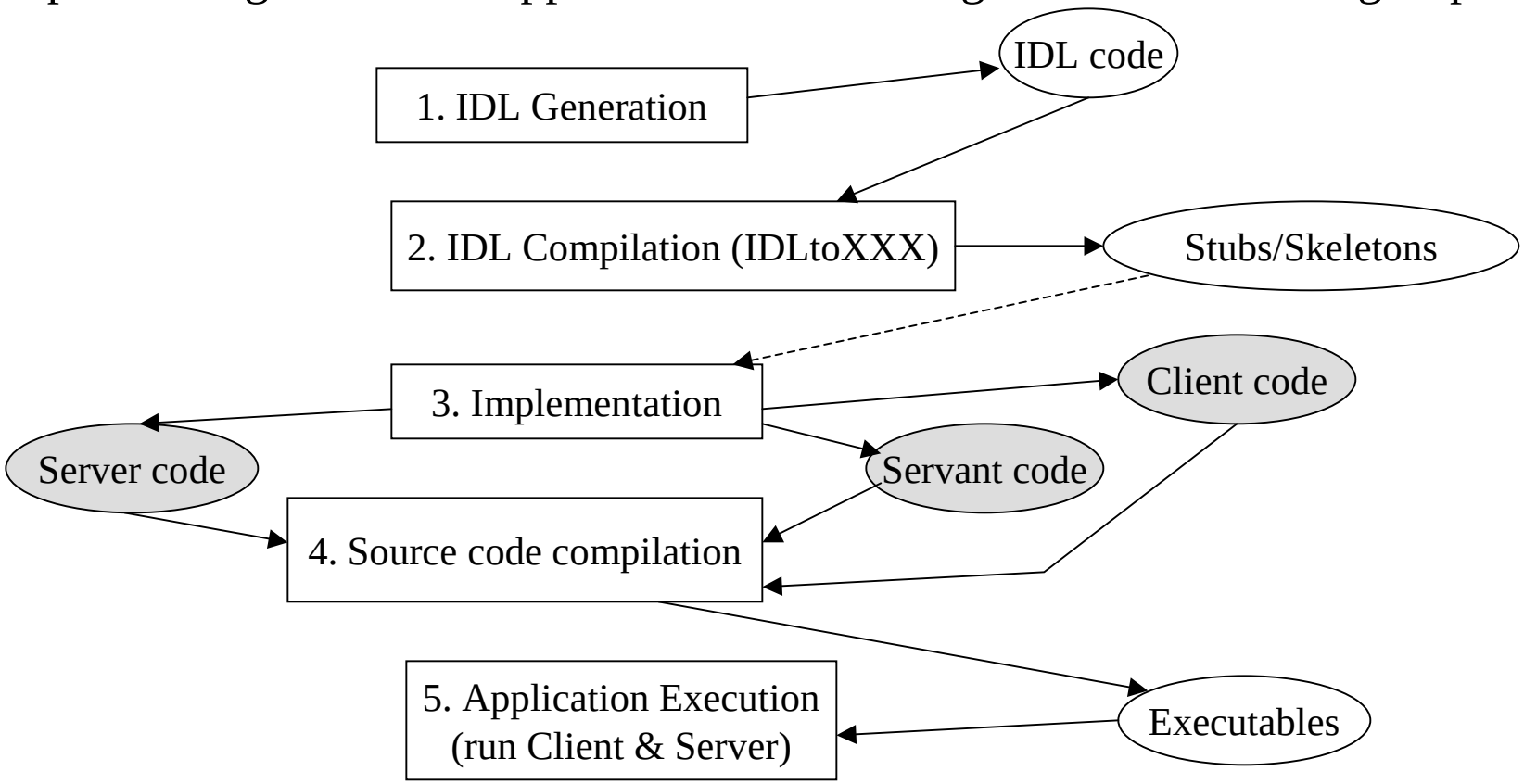
String name ="Calc";

NameComponent path[] = ncRef.to_name(name);

ncRef.rebind(path,fref);

3. Implementation: General Approach

-Implementing a CORBA application involves in general the following steps:



÷The Servant corresponds to the remote CORBA object class

÷The Client and Server may either be implemented in the same language or in different languages (e.g., C++ & Java). CORBA applications can interact, regardless of the language they are written in.

4. Java Implementations

IDL Code

Example:

•We define an IDL file, named *calc.idl* that describes a CORBA service *Functions* (encapsulated in a package or library called Calculator) that provides two operations, *square_root* and *power*.

```
module Calculator {

        interface Functions {
                float square_root (in float number);
                float power (in float base, in float exponent);
        };

};
```

Compiling IDL files

idlj -fall calc.idl

For every construct in the IDL file that maps to a Java class or interface, a separate class file is generated. Directories are automatically created for those IDL constructs that map to a Java package (e.g., a module).

This command generates several Java source files on which the actual implementation will be based:

- Functions.java
- FunctionsHelper.java
- FunctionsHolder.java
- FunctionsOperations.java
- FunctionsPOA.java
- _FunctionsStub.java

3. Compile the generated classes

javac Calculator*.java

- Recommended that the physical location of the classes you create remain separate from the generated classes.
- For instance one directory level above the *Calculator* subdirectory (which contains the generated classes).

Servant Code

•The servant class *FunctionsImpl* must inherit from the generated class *FunctionsPOA*.

```
import Calculator.*;
//First, extend the implementation Base class
public class FunctionsImpl extends FunctionsPOA {


        private ORB orb;
        public void setORB(ORB orb_val) {
                orb = orb_val;
        }


//A constructor is not required, but is recommended
        public FunctionsImpl() {
        }
//Implement the two special methods
        public float square_root (float number) {
                return (float) Math.sqrt ((double) number);
        }
        public float power (float base, float exponent) {
                return (float)Math.pow ((double)base,(double)exponent);
        }
}
```

Server Code

•The Server class holds the server's main() method:

•The server:

- •Creates and initializes an ORB instance
- •Creates a servant instance
- •Gets a reference to the root POA
- •Activates the POA's POAManager
- •Associates the servant with the POA
- •Gets a CORBA object reference for the root naming context in which to register the new CORBA object
- •Narrows the object reference to a naming context
- •Registers the new object in the naming context
- •Waits for invocations of the new object from the client

•The Calculator Server class:

```
import Calculator.*; //package containing the stubs and skeletons
import org.omg.CosNaming.*; //naming service
/*special exceptions thrown by naming services*/
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*; //all corba applications need these classes
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;
import java.util.Properties

public class CalculatorServer {

  public static void main(String args[]) {

    try{
        //create and initialize an instance of a server-side ORB
          ORB orb=ORB.init(args,null);

          //get reference to  rootpoa & activate the POAManager
          POA rootpoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
          rootpoa.the_POAManager().activate();
```

//create servant (implementation object) and register it with the ORB

```
FunctionsImpl f= new FunctionsImpl();
f.setORB(orb);
```

//get object reference from the servant

```
org.omg.CORBA.Object ref = rootpoa.servant_to_reference(f);
Functions fref = FunctionsHelper.narrow(ref);
```

//get the root naming context; use NamingContextExt instead of
//NamingContext-is part of the interoperable naming service (INS).

```
org.omg.CORBA.Object objRef=orb.resolve_initial_references("NameService");
NamingContextExt ncRef=NamingContextExtHelper.narrow(objRef);
```

//bind the Object Reference in Naming

```
String name ="Calc";
NameComponent path[] = ncRef.to_name(name);
ncRef.rebind(path,fref);
```

//wait for invocations from clients

```
orb.run();
    } catch (Exception e) {
        System.err.println("Error: " + e);
        e.printStackTrace();
       }
    }
 }
```

Client Code

•The client application will locate a reference to the Functions object (i.e., “Calc”) using the Naming service.

Calculator Client:

```
import Calculator.*; //package containing the stubs
import org.omg.CosNaming.*; // naming service
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*; //all corba applications need these classes

public class CalculatorClient {
  static Functions fImpl;
  public static void main(String args[]) {
    try{
```

//create and initialize an instance of a client-side ORB

```
ORB orb=ORB.init(args,null);
```

//get the root naming context; use NamingContextExt instead of //NamingContext-is part of the interoperable naming service (INS).

```
org.omg.CORBA.Object objRef= orb.resolve_initial_references(“NameService”);
NamingContextExt ncRef=NamingContextExtHelper.narrow(objRef);
```

```
//Look up the object bound to the name "Calc". Use the Helper class to
// "cast" the generic CORBA object reference to a Functions implementation.
        String name = "Calc";
        fImpl = FunctionsHelper.narrow(ncRef.resolve_str(name));

// The object returned by the narrow method is actually
// a _FunctionsStub object that implements the methods in the
// Functions interface

//Use the reference to execute the interface methods
        float sqrt fimpl.square_root (10f);
        float pow=fimpl.power (2f, 8f);

        System.out.println("The square root of 10 is: " +sqrt);
        System.out.println("2 to the 8th power is: " +pow);

        } catch (Exception e) {
                System.out.println("ERROR : " + e);
                e.printStackTrace();
        }
    }
}
```

Compiling and Running the Application

1. Compile the implementation, client and server code

javac –d . FunctionsImpl.java CalculatorServer.java CalculatorClient.java

2. Start ORB daemon, orbd, which includes a naming service, on the server machine.

•The default port number is 900; here we change to 1050

start orbd -ORBInitialPort 1050 –ORBInitialHost servermachinename

(from Unix: orbd -ORBInitialPort 1050 –ORBInitialHost servermachinename)

3. Start the server

java CalculatorServer -ORBInitialPort 1050

(*ORBINitialHost* is omitted because the name server is running on the same machine as the server)

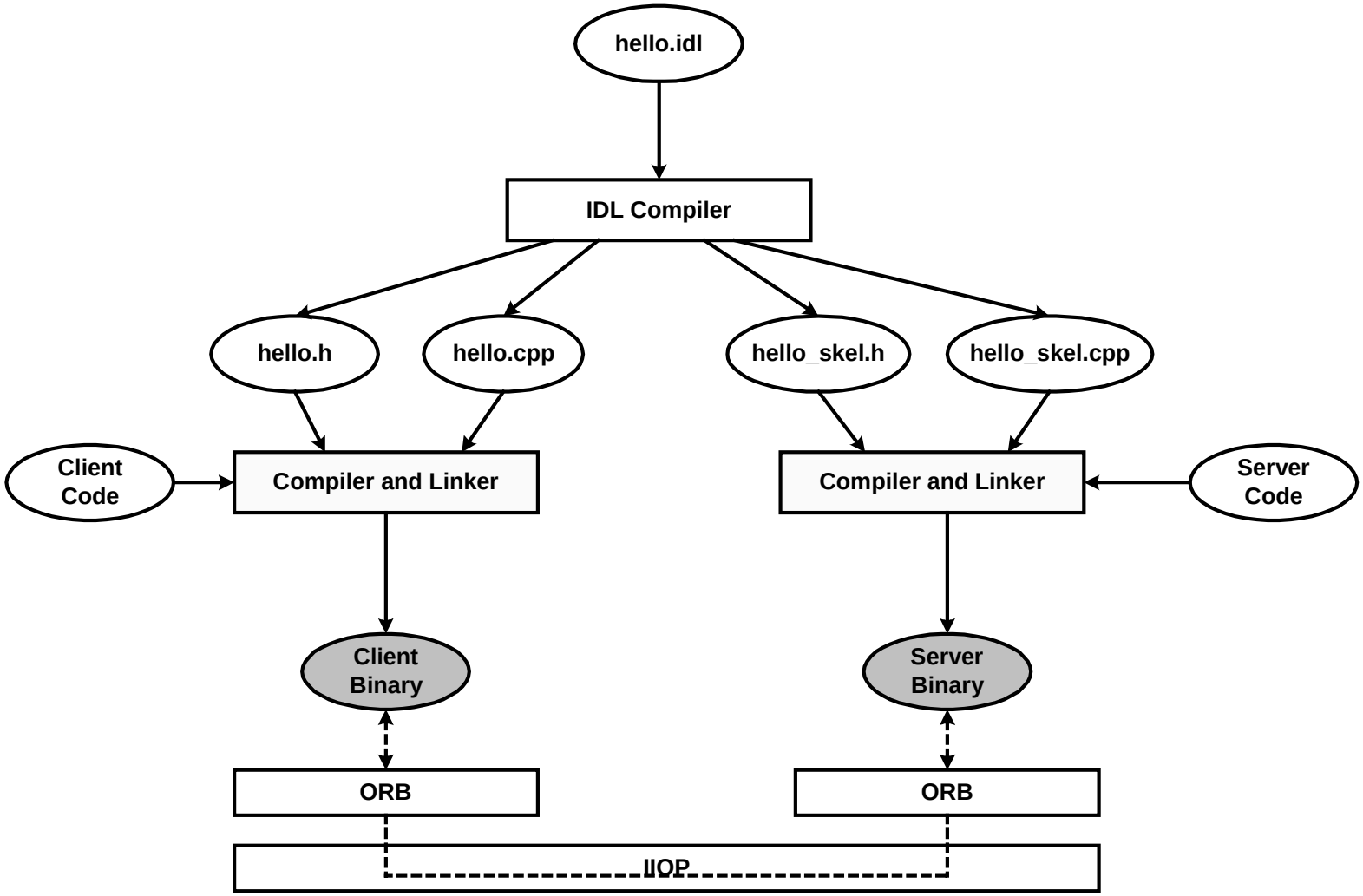
4. Start the client

java CalculatorClient -ORBInitialHost nameserverhost -ORBInitialPort 1050

(*nameserverhost* is the host on which the IDL server is running. In this case it the server machine; in any case, you'll start only one name server for both the client and server).

5. C++ Implementation

IDL to C++ Compiler Functionality



Example of C++ Implementations

IDL Code

```
// Hello.idl
interface Hello {
        void say_hello();
};
```

Compiling the IDL

> idl Hello.idl

This command will create the files:

- Hello.h : Header file containing Hello.idl ’s translated data types and interface stubs
- Hello.cpp: Source file containing Hello.idl ’s translated data types and interface stubs
- Hello_skel.h: Header file containing skeletons for Hello.idl ’s interfaces
- Hello_skel.cpp: Source file containing skeletons for Hello.idl ’s interfaces

Servant Definition and Implementation

To implement the server, we need to define an implementation class for the Hello interface. To do this, we create a class Hello_impl that is derived from the “skeleton” class POA_Hello , defined in the file Hello_skel.h .

//Hello_impl.h : servant definition

```
#include <Hello_skel.h> //contains the skeleton class POA_Hello

class Hello_impl : public POA_Hello, public PortableServer::RefCountServantBase {
                        public:
                            virtual void say_hello() throw(CORBA::SystemException);
};
```

//Hello_impl.cpp: servant implementation

```
#include <iostream.h>
#include <OB/CORBA.h> //contains definitions for the standard CORBA classes
#include <Hello_impl.h>

void Hello_impl::say_hello() throw(CORBA::SystemException) {
            cout << "Hello World!" << endl;
}
```

Server Program

To simplify exception handling and ORB destruction, we split the server into two functions: main() and run() , where main() only creates the ORB, and calls run()

//Server.cpp

```
#include <OB/CORBA.h>
#include <Hello_impl.h>
#include <fstream.h>
int run(CORBA::ORB_ptr);  // Forward declaration for the run() function.

int main(int argc, char* argv[]) {
        int status = EXIT_SUCCESS; // Exit status
        CORBA::ORB_var orb;
        try{ //initialize the ORB using the parameters with which the program was started.
                orb = CORBA::ORB_init(argc, argv);
                status = run(orb);}
        catch (const CORBA::Exception&) {status = EXIT_FAILURE;}

        if(!CORBA::is_nil(orb)) { //If the ORB was successfully created, destroy it to free resources.
                try {
                        orb -> destroy();
                }
                catch(const CORBA::Exception&) {status = EXIT_FAILURE; }
        }
        return status; //If there was no error, EXIT_SUCCESS is returned,or EXIT_FAILURE otherwise
}
```

//Server.cpp (ctd.)

```
int run(CORBA::ORB_ptr orb) {

        //Use the ORB reference to obtain a reference to the Root POA.
        CORBA::Object_var poaObj = orb -> resolve_initial_references("RootPOA");
        PortableServer::POA_var rootPoa = PortableServer::POA::_narrow(poaObj);

         //Use the Root POA to obtain a reference to its POA Manager.
        PortableServer::POAManager_var manager = rootPoa -> the_POAManager();
```

/*Create and assign a servant object to a ServantBase_var variable. The servant is then used to incarnate a CORBA object, using the _this() operation. ServantBase_var and Hello_var , like all _var types, are "smart" pointer, i.e., servant and hello will release their assigned object automatically when they go out of scope.*/

```
        Hello_impl* helloImpl = new Hello_impl();
        PortableServer::ServantBase_var servant = helloImpl;
        Hello_var hello = helloImpl -> _this();
```

*//Server.cpp-*int run(CORBA::ORB_ptr orb) (ctd.)

/*The client must be able to access the implementation object. This can be done by saving a “stringified” object reference to a file, which can then be read by the client and converted back to the actual object reference.*/

```
CORBA::String_var s = orb -> object_to_string(hello);
const char* refFile = "Hello.ref";
ofstream out(refFile);
out << s << endl;
out.close();
```

/*The server must activate the POA Manager to allow the Root POA to start processing requests, and then inform the ORB that it is ready to accept requests.*/

```
manager -> activate();
orb -> run();

return EXIT_SUCCESS;
}
```

Client Implementation

Several segments of the client program are similar to the server program; for instance, the code to initialize and destroy the ORB is the same.

//Client.cpp

```
#include <OB/CORBA.h>
#include <Hello.h> //In contrast to the server, the client does not need to include Hello_impl.h .
                   //Only the generated file Hello.h is needed.
#include <fstream.h>

int run(CORBA::ORB_ptr);

int main(int argc, char* argv[]) {
        ... // Same as for the server
}
```

//Client.cpp (ctd.)

```
int run(CORBA::ORB_ptr orb) {

/* The “stringified” object reference written by the server is read and converted to a CORBA::Object
   object reference. It’s not necessary to obtain a reference to the Root POA or its POA Manager,
   because they are only needed by server applications.*/

        const char* refFile = "Hello.ref";
        ifstream in(refFile);
        char s[2048];
        in >> s;
        CORBA::Object_var obj = orb -> string_to_object(s);

       //Generates a Hello object reference from theCORBA::Object object reference.
        Hello_var hello = Hello::_narrow(obj);

       //Invoke the say_hello operation on the hello object reference
        hello -> say_hello();

        return 0;
}
```

Compiling and Linking

-Compiling Hello.cpp results in an object file with the following name:
- **UNI**X: Hello.o
- **Window**s: Hello.obj

-You must link both the client and the server with the file for your platform. The compiled Hello_skel.cpp and Hello_impl.cpp are only needed by the server.

-In practice, compiling and linking is compiler- and platform-dependent. Many compilers require unique options to generate correct code.

-To build Orbacus programs, you must at least link with the Orbacus library for your platform:
- **UNIX**: libOB.a
- **Window**s: ob.lib

Running the Application

-The “Hello World!” application consists of two parts:
- The client program
- The server program

-Start the server first, since it must create the file Hello.ref that the client needs in order to connect to the server. As soon as the server is running, you can start the client.

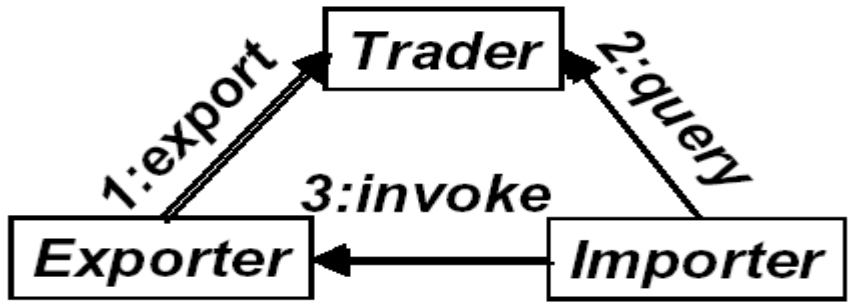
7. CORBA Trading Service

Rationale

- Locating objects in location transparent way
- Naming simple but may not be suitable when
  - clients do not know server
  - there are multiple servers to choose from
- Trading supports locating servers based on service functionality and quality
- Naming ⇔ White pages
- Trading ⇔Yellow Pages

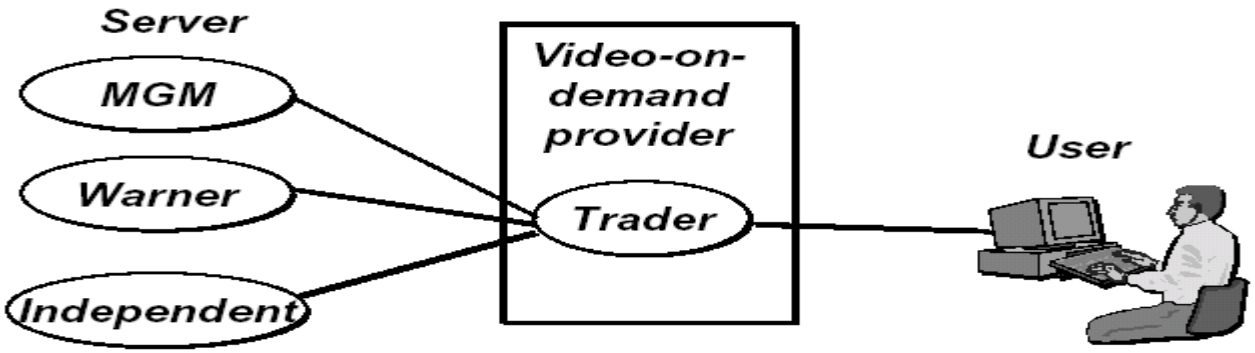
Trading Operation

- Trader operates as broker between client and server.
- Enables client to change perspective from ´who?´ to ´what?´
- Similar ideas in:
  - mortgage broker
  - insurance broker

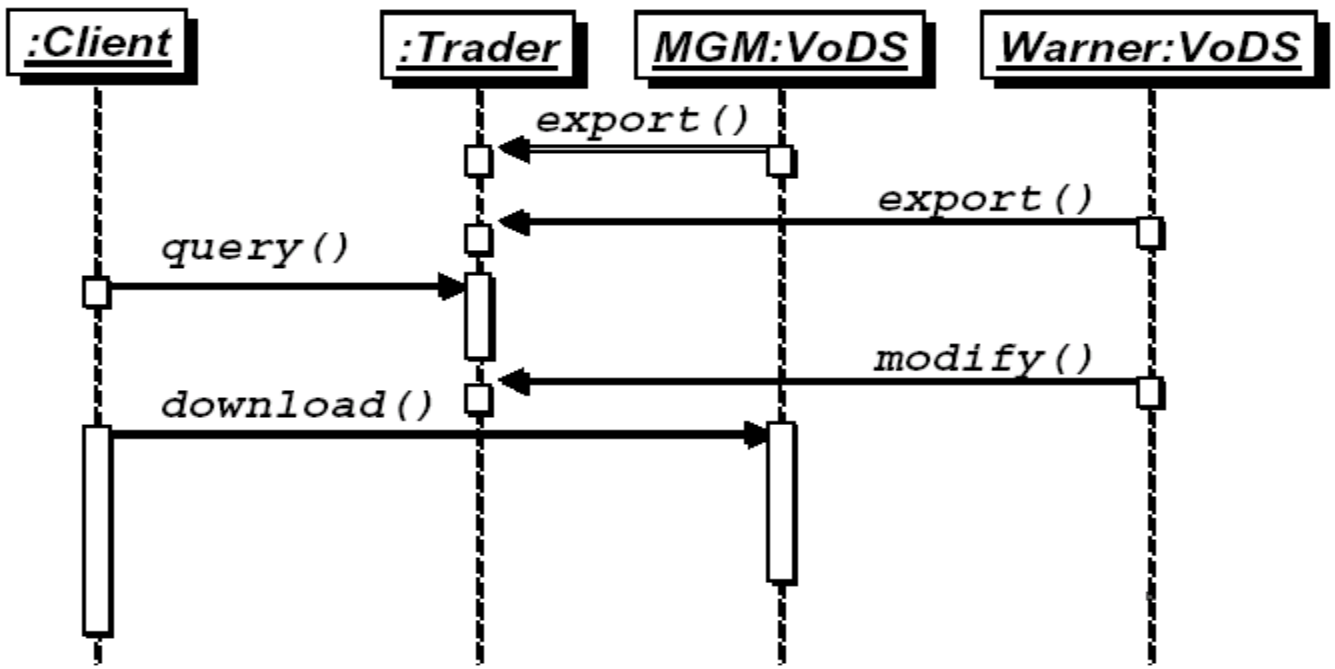


- Clients ask trader for
  - a service of a certain type
  - at a certain level of quality
- Trader supports
  - service matching
  - service shopping
- Server registers service with trader.
- Server defines assured quality of service:
  - Static QoS definition
  - Dynamic QoS definition

Example: Hongkong Telecom video-on-demand



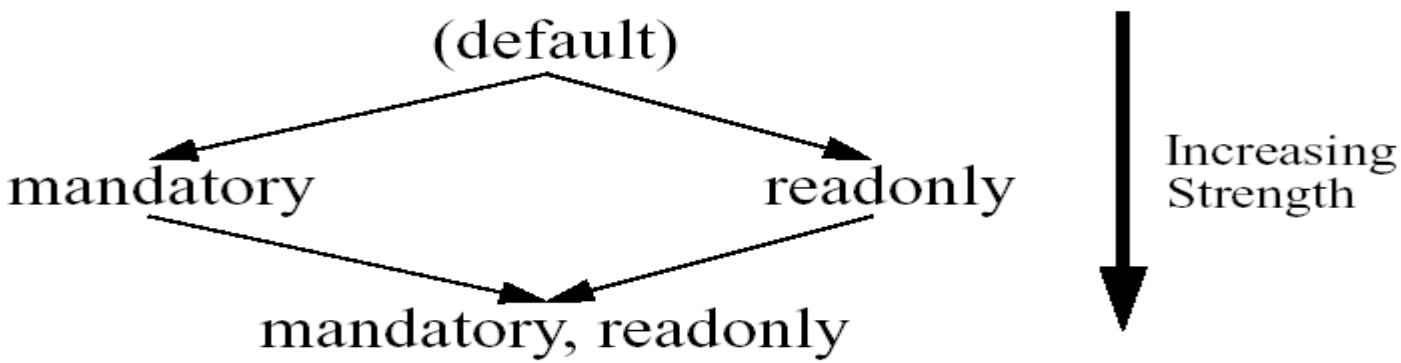
The Trading Process



Service Type Definition

- Service types define
  - Functionality provided by a service and
  - Qualities of Service (QoS) provision.
- Functionality defined by object type
- QoS defined based on properties, i.e.
  - property name
  - property type
  - property value
  - property mode
    - mandatory/optional
    - readonly/modifiable

Property Strength



Service Type Example

```
typedef enum {VGA,SVGA,XGA} Resolution;
service video_on_demand {
        interface VideoServer;
        readonly mandatory property float fee;
        readonly mandatory property Resolution res;
        modifiable optional property float bandwidth;
}
```

Service Type Hierarchy

- An object type might have several implementations with different QoS
- Same object type might be used in different service types.
- Service type S is subtype of service S' iff
  - –object type of S is identical or subtype of object type of S'
  - –S has at least all properties defined for S'
- Subtype relationship can be exploited by trader for service matching purposes

Constraint Definition

- Importer defines the desired qualities of service as part of the query:
  - *Example:*

```
fee<10 AND res >=SGA AND bandwidth>=256
```

- In a query, trader matches only those offers that meet the constraint

Federated Traders

- Scalability demands federation of traders
- A trader participating in a federation
  - offers the services it knows about to other traders
  - forwards queries it cannot satisfy to other traders
- Problems
  - Non-termination of import
  - Duplication of matched offers

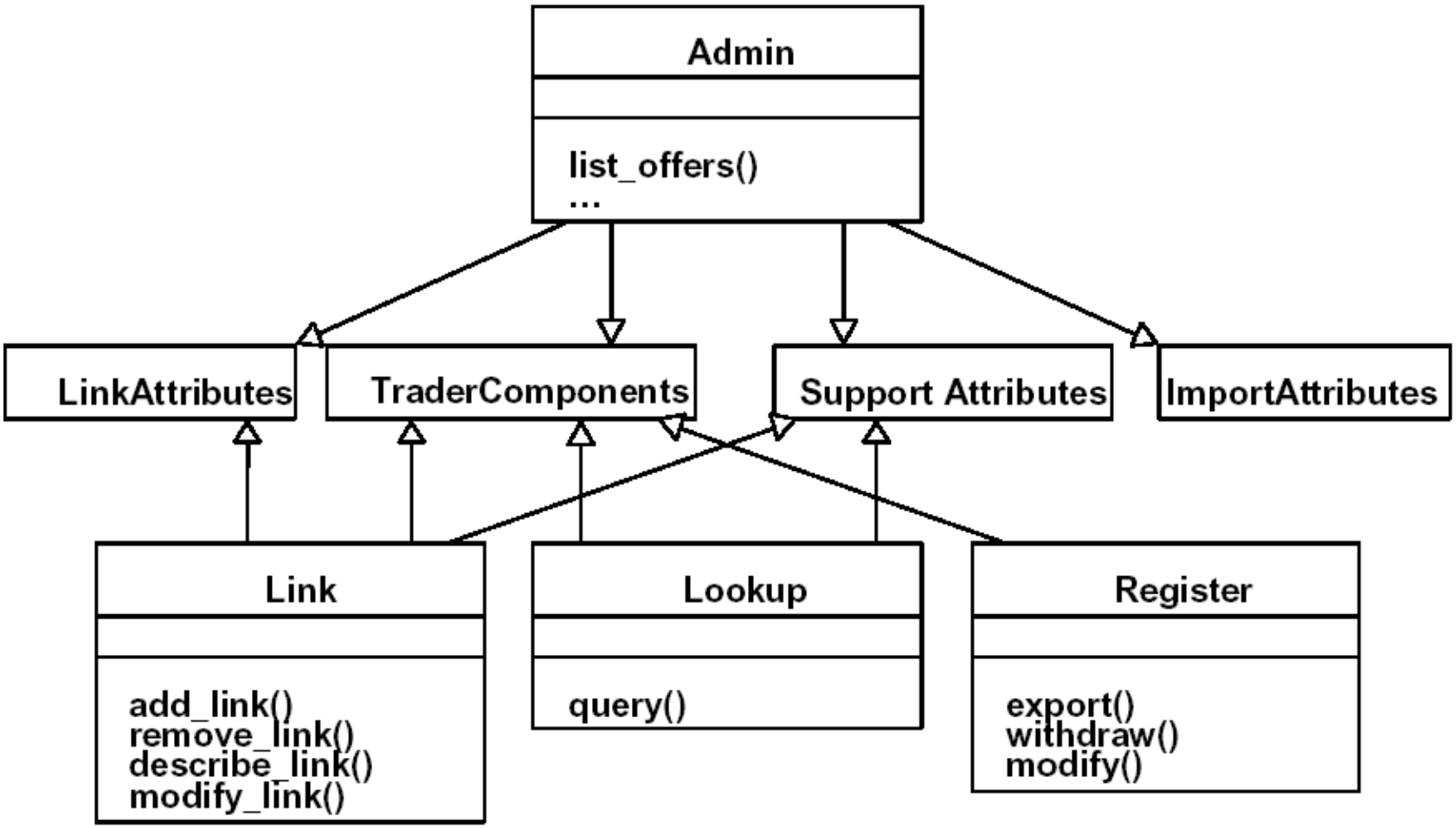
Trading Policies

- Depending on constraint and available services, a large set of offer might be returned by a query.
- Trading policies are used to restrict the size of the matched offers
  - Specification of an upper limit
  - Restriction on service replacements
  - Restriction on modifiable properties (these might change between match making and service requests)
- Policies provide information to affect trader behavior at run time.

- Policies are represented as name value pairs.

```
typedef string PolicyName; // policy names restricted to Latin1
typedef sequence<PolicyName> PolicyNameSeq;
typedef any PolicyValue;
struct Policy {
        PolicyName name;
        PolicyValue value;
};
typedef sequence<Policy> PolicySeq;
```

CORBA Trading Interfaces



Defining Quality of Service

```
typedef Istring PropertyName;
typedef sequence<PropertyName> PropertyNameSeq;
typedef any PropertyValue;
struct Property {
                PropertyName name;
                PropertyValue value;
};
typedef sequence<Property> PropertySeq;
enum HowManyProps {none, some, all}
union SpecifiedProps switch (HowManyProps) {
        case some : PropertyNameSeq prop_names;
};
```

Properties are <name, value> pairs. An exporter asserts values for properties of the service it is advertising. An importer can obtain these values about a service and constrain its search for appropriate offers based on the property values associated with such offers.

Trader Interface for Exporters

```
interface Register {
OfferId export(in Object reference,
               in ServiceTypeName type,
               in PropertySeq properties) raises(...);
OfferId withdraw(in OfferId id) raises(...);
void modify(in OfferId id,
                        in PropertyNameSeq del_list,
                        in PropertySeq modify_list) raises (...);
};
```

Service Offers

- A service offer is the information asserted by an exporter about the service it is advertising. It contains:
  - –the service type name,
  - –a reference to the interface that provides the service, and
  - –zero or more property values for the service.
- An exporter must specify a value for all mandatory properties specified in the associated service type.
  - In addition, an exporter can nominate values for named properties that are not specified in the service type. In such case, the trader is not obliged to do property type checking.

```
struct Offer {
                    Object reference;
                    PropertySeq properties;
};
typedef sequence<Offer> OfferSeq;

struct OfferInfo {
                    Object reference;
                    ServiceTypeName type;
                    PropertySeq properties;
};
```

Trader Interface for Importers

```
interface Lookup {
void query(      in ServiceTypeName type,
                        in Constraint const,
                        in Preference pref,
                        in PolicySeq policies,
                        in SpecifiedProps desired_props,
                        in unsigned long how_many,
                        out OfferSeq offers,
                        out OfferIterator offer_itr,
                        out PolicyNameSeq Limits_applied)
raises (IllegalServiceType, UnknownServiceType, IllegalConstraint, IllegalPreference,IllegalPolicyName, PolicyTypeMismatch,
          InvalidPolicyValue, IllegalPropertyName, DuplicatePropertyName, DuplicatePolicyName );
};
```