Programming with Spring

Outline

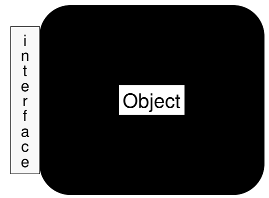
- ❏ Basics
- ❏ Name Servers
- ❏ Interfaces
- ❏ Inheritance
- ❏ Using Spring Objects

Explanation:

In this module we introduce the Spring notion of objects and start programming with them.

Basics

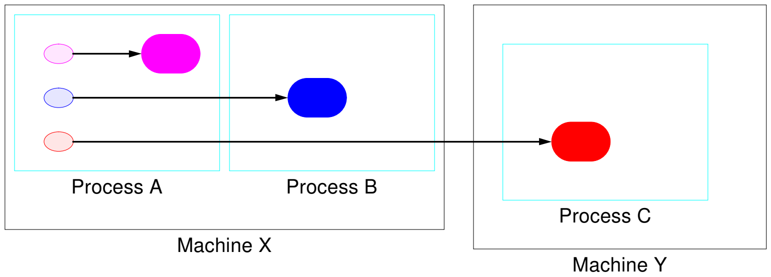
It's the Abstraction That's Important



Explanation:

Spring objects are fully *abstract*—all that a user of an object can find out about the object is its public interface; no information about its internals is available to the outside world. This makes it possible for users of an object to be completely separated from its implementation, an important concern in distributed computing. What's more, multiple implementations of an object can all share the same interface.

Location Transparency

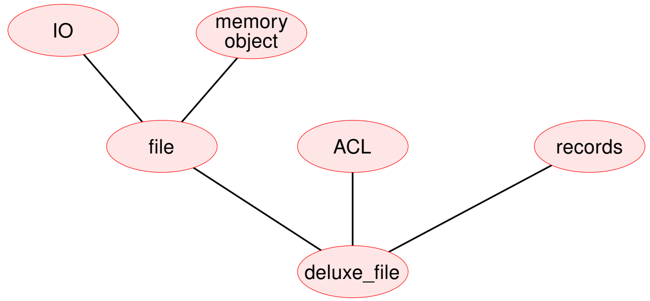


Explanation:

An important aspect of our object model is that methods are referred to and invoked on objects in exactly the same way, no matter where the objects are. In the picture, the small ovals are *object references* and the rectangles with rounded corners are objects. The user (or *client*) of an object uses the object reference as a means for invoking methods on the object. From the user's perspective, it makes no difference whether the object is in the same address space as the client, in a different address space but on the same machine as the client, or on a different machine from the client.

Clearly, there are differences in how the runtime handles the three situations shown in the picture. The invocation of a method on a local object can (and does) take advantage of optimizations that aren't available for remote objects.

Interface Inheritance



Explanation:

Most approaches to object-oriented computing make use of what is known as *implementation inheritance*. For example, in C++, the declaration of a class can specify that the class inherits from other classes. The picture shows an inheritance hierarchy in which *deluxe_file* inherits from *file*, *ACL*, and *records*. *File* inherits from *IO* and *memory object*. In C++, a declaration of the *deluxe_file* class would be based on the following:

```
class deluxe_file :
        public file, public ACL, public records
{
public:
    ...
private:
    ...
};
```

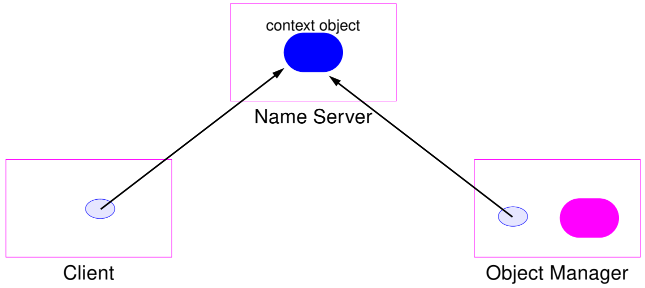
Thus all of the methods of *file*, *ACL*, and *records* are available in objects of type *deluxe_file*. Moreover, the implementation of these methods are inherited in *deluxe_file*: the code for *deluxe_file* includes the code of the classes it inherits. This is a useful concept for C++. One can save a lot of time by reusing code, and this notion of implementation inheritance makes reusing code easy.

However, implementation inheritance is not ideal in all situations. In the example in the slide, *ACL* is a class that provides an interface for manipulating access-control lists. Thus, since *deluxe_file* inherits from *ACL*, *deluxe_file* objects can be treated as access-control lists—one can invoke ACL-specific methods on files as if they were purely access-control lists. But, even though we have a standard interface for operating on ACLs, we do not want to require that all ACLs be implemented in the same way. Different file systems might use different implementations: ACLs might be used in a database whose implementation of ACLs might bear little resemblance to the ACL implementations of file systems.

The ability to specify interfaces that are common to a number of different classes is very important. But it would be a serious problem if we were then forced to use the same implementation for all uses of the interface.

Note that implementation inheritance is not ruled out: Spring's notion of interface inheritance permits implementation inheritance but does not require it.

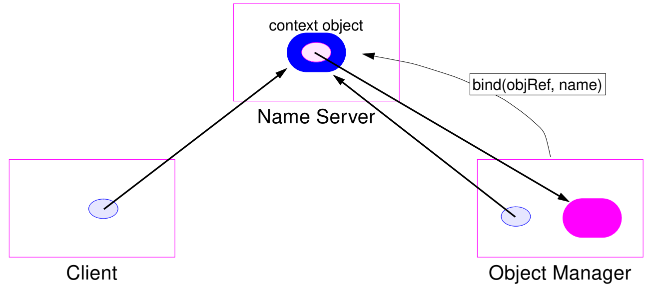
Name Servers



Explanation:

A *Name Server* is an agent that maps names to object references. It provides *contexts*, which are analogous to the directories of file systems.

Binding

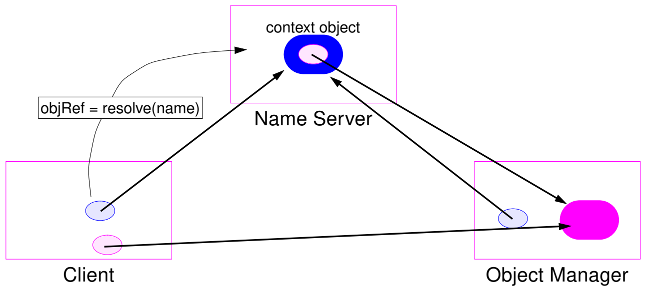


Explanation:

Here the object manager invokes the context object's *bind* method to *bind* a reference to its object to a name and put this binding into the context object.

Name Servers

Resolving

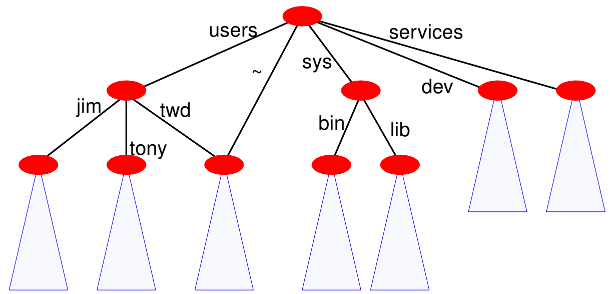


Explanation:

The client now obtains a copy of the object reference by *resolving* the name from the context object (for which it has a reference).

Name Servers

Process Name Space

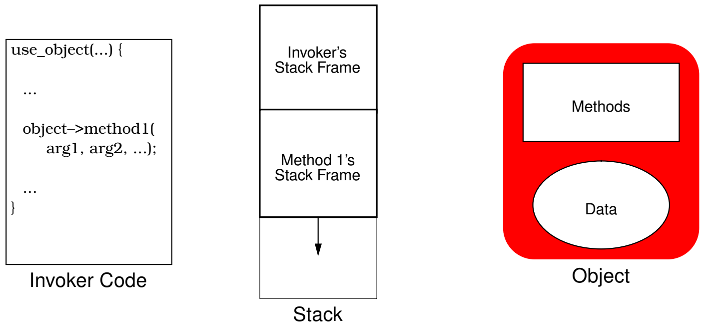


Explanation:

A number of name servers provide the complete Spring name space, each managing a portion of it. Each process has its own name space, which has both private and global portions.

Interfaces

Local Objects

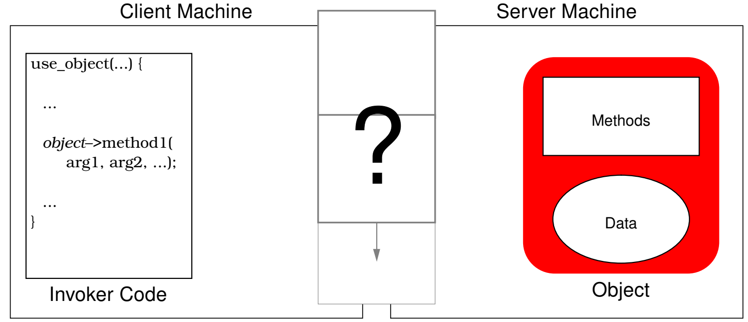


Explanation:

Let's consider what happens when one invokes a method on a local object. Associated with the current thread is a stack. The caller has a stack frame in which it puts, among other things, the arguments for the method invocation. As part of the method invocation, a new stack frame is pushed on the stack. The code for the method reaches up into the caller's stack frame to find its arguments.

Interfaces

Remote Objects



Explanation:

Now consider invoking a method on a remote object. If we try to set up the stack as in the local case, we run into a problem: the invoker might put the arguments in its stack frame, but how does the remote code of the method access these arguments?

Interfaces

Concerns

- ❑ Separation of *interface* and *implementation*
  - – *Contract* (*IDL* specification)
- ❑ Managing the interaction between invoker and object
  - – Subcontract

Explanation:

What's needed are some intermediaries, known as *stubs*, to handle the communication of control and data. The client-side stub presents an interface that looks like that of the actual object, i.e., the client-side stub appears to be the actual object. Instead, however, its methods contain code to contact the server. On the server is a server-side stub routine that receives requests from the client-side stub and converts them into an invocation of the actual object. On return from the object's method, control goes back to the server-side stub, which communicates the results back to the client-side stub, which finally returns the results back to the original invoker.

There are two issues we need to deal with:

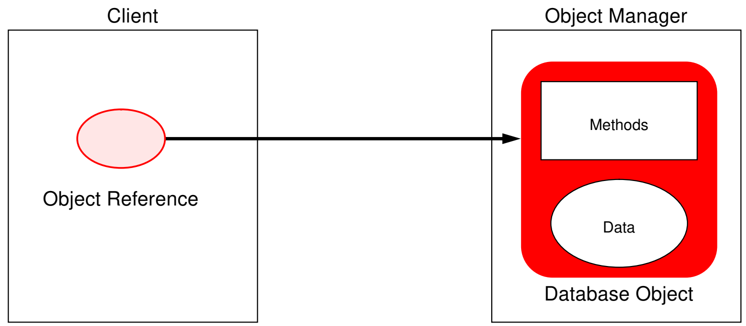
- how do we produce the stubs?
- what do the stubs actually do?

The answer to the first question is that we make stub generation as automatic as possible. To produce a stub, all that is required is a specification of the interface provided by the object. This specification, or *contract*, is written in *IDL*—interface definition language. Given such a specification, the necessary stubs and associated declarations and definitions are produced by the IDL compiler. This approach has the additional benefit of keeping separate the description of an object's interface from the object's implementation. Thus the implementation can change without requiring any changes in the clients of the object.

The other issue is what is going on inside the stub. Here too we split the problem into two pieces. What is actually generated by the IDL compiler is code that makes use of another layer, the *subcontract*, to do the actual work. The subcontract handles communication, the marshaling and unmarshaling of objects, and anything else that might be required to handle the interaction between client and server. A number of subcontracts are supplied with Spring; they all provide the same interface to the IDL-compiler-generated code, so substitution is easy.

Interfaces

An Example



Explanation:

As an example, we look at an object manager that handles a simple database. The client has a reference to this object, as shown in the slide. In the framework of Spring objects this reference is (at least conceptually) a pointer. However, when we map this onto a programming language such as C++, it becomes something a bit more complicated.

Interface Vs. Declaration

```
// Contract (in IDL)

interface NVcollection {
 string query(copy string name);
 boolean add( copy string name,
              copy string value);
 boolean remove(copy string name);
}
```

```
// Declaracotion (in C++)

class NVcollection {
public:
  string query(string name);
  bool add(string name, string value);
  bool remove(string name);

  ...
private:
  // this is really none of your business
  type1 func1(type2 arg1, type3 arg2);
  type4 func2(type5 arg1);
  ...
  type146 root;
}
```

Explanation:

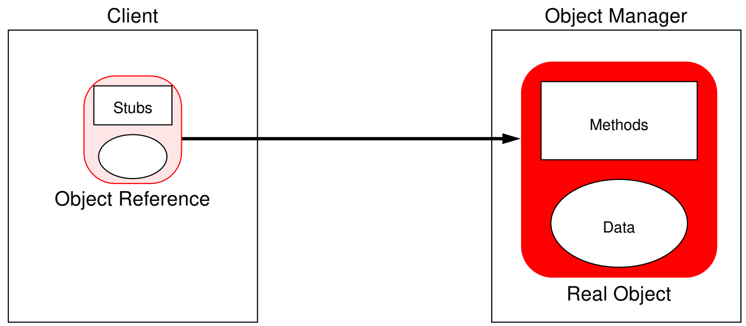
A Spring object presents an implementation as specified in IDL. Here we give the *NVcollection* interface, which supplies three methods. Note that, besides identifying the types of the parameters to the methods, we also specify the parameter-passing *modes*. We discuss these in more detail later, but here we must supply instructions for how the parameters should be transferred. We are using *copy*, which specifies that a copy of the parameter is to be passed from the client to the server. The implicit mode for the return value of these methods is *produce*, which means that the return value is transferred (without copying) from the server back to the client.

We must, of course, represent our Spring objects in terms of some programming language. There is no architecturally preferred language; in principle any programming language could be used, whether object-oriented or not. In practice what has been used is strictly C++. Though it is object-oriented, the C++ notion of an object is not the same as that of Spring. We therefore must use various conventions for representing Spring objects and object references in C++.

The slide shows part of the representation of a Spring object in C++. The important thing to note here is that the declaration of the object in IDL makes no mention of any internal details—such information is not just hidden, it's not there at all. The C++ version of the declaration (which is produced with the aid of the IDL compiler) must provide full detail. This detail is essential on the server but irrelevant on the client, which only needs the level of detail provided by the IDL specification.

This separation of interface from implementation, even to the extent that the implementation is described in a C++ declaration, is of crucial importance in Spring. It allows the complete separation of client from server; any change whatsoever can be made to the implementation of an object, as long as the implementation remains faithful to the specification of its interface.

The Contract



Explanation:

A reference to a remote object can't be simply a pointer to the object—the client and object are in different address spaces. What must happen when a client invokes a method on a remote object via the object reference is that client-side stub code is invoked, which in turn marshals (or causes to be marshaled) the outgoing data, places the call, receives the response, unmarshals the incoming data, and finally returns to the caller.

Thus, while in concept an object reference is merely a pointer, in practice it is a C++ object itself (or something equivalent if another programming language is being used) that provides public methods corresponding to the methods of the Spring object, but whose implementation forms the stub.

Parameter-Passing Modes

- Consume
- *Produce* (a.k.a. *out*)
- *Copy* (a.k.a. *in*)
- Borrow

Explanation:

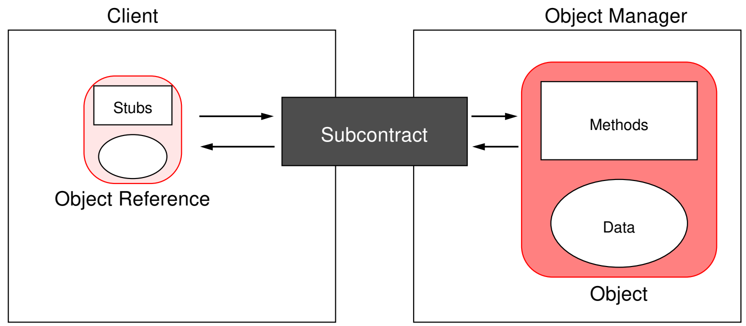
The slide lists the four parameter-passing modes of Spring. *Consume* is used when the caller has no need for the parameter after the call, and thus it can be transferred to the callee. This means that no copying is necessary and simplifies the transfer of data. *Produce* is the analogous mode, but for the opposite direction of transfer. (Functional return values are handled as *produce* parameters.) *Consume* and *produce* have been found useful in a large number of cases, so that the optimizations they allow are quite beneficial.

Copy parameters are copied on the client side, then transferred to the server. Besides the overhead required for the copy of data, copying object references can be especially expensive, because some subcontracts require that the server be notified every time a new reference is made to one of the server's objects. Thus passing such parameters could require an extra remote procedure call to the server.

Borrow is used in two circumstances: one is where the caller does not need the argument for the duration of the call, but does require it after the call has completed. This certainly requires care in multithreaded callers, but it can save the overhead of a copy. The other use of *borrow* is what is often called *in-out*: one item is transmitted (without copying) and another item is returned in the original's place.

Interfaces

The Subcontract



Explanation:

The subcontract, i.e., the code that handles the communication with the object manager and marshals object references, is a layer in itself. We discuss subcontracts in more detail in later modules; note, however, that any issues involving efficiency of data transfer, replication, caching, etc., are dealt with at this layer and are independent of the specification and implementation of the object.

Steps

- Client
  1) Resolve object reference from name
  2) Invoke method on object

- Server
  1) Create object
  2) Bind object to name
  3) Receive method invocation

Explanation:

Now that we are familiar with Spring objects, we look at the details of programming with them. The slide shows a very high-level outline of a client and a server. Of course, the server's steps would be executed before the client's, but we discuss the client's steps first.

Complete IDL Specification of the Contract

```
// dbex.idl

module dbex {
    interface NVcollection {
      string query(copy string name);
      boolean add(copy string name,
                  copy string value);
      boolean remove(copy string name);
    };
};
```

Explanation:

We continue with the example begun on page 12. We are specifying a simple database that associates strings with names (we call it a "name-value collection"). Our interface provides three methods: one to query the database, one to add name-value pairs, and one to remove name-value pairs. To guard against naming conflicts, the interface is put inside of another construct, the *module*, which groups together any number of interfaces.

Keep in mind that this is merely an illustrative example (the functionality it provides is subsumed by the name service).

Specifying the Subcontract

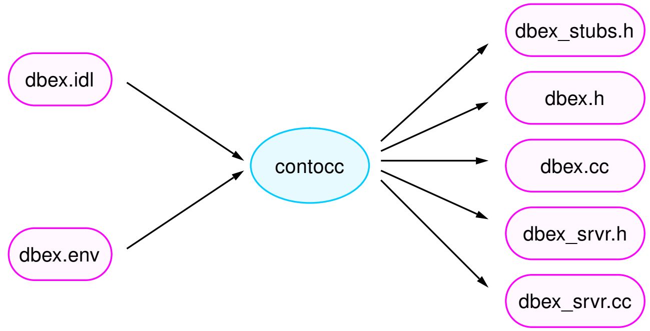
```
// dbex.env

$singleton: sc=singleton
            sc_file="spring/subcontracts/singleton.h";
dbex::NVcollection:  $singleton;
```

Explanation:

Normally, we do not have to worry about specifying the subcontract—the default is the *singleton* subcontract, which is what we usually want. If we want to specify some other subcontract or to be explicit about the singleton subcontract, we can supply an *environment file* that indicates the subcontracts for selected interfaces, as shown on the slide.

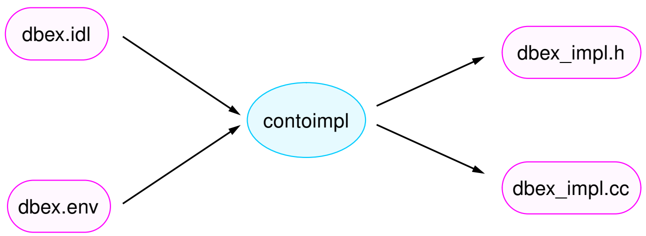
Contocc



Explanation:

The IDL specification is a complete specification of the interface to a Spring object. However, to use this specification, we must convert it into something meaningful in whatever programming language we are using. Assuming that our language is C++, we must produce client-side and server-side stub code (see page 11) for use with C++ programs. (Of course, if we plan to use a number of programming languages, we could produce client- and server-side stubs in each; a client written in one language would have no problem communicating with a server written in another language.) The *contocc* program produces, from the IDL specification and, if supplied, the environment file, a set of files that together comprise the client- and server-side stub code. In the next several pages, we proceed with our example to see how these files are used.

Contoimpl



Explanation:

Contocc produced the stub code. On the server side, the stub code forms a base class that is inherited by the application code (see page 24). To set up the appropriate declarations to establish this hierarchy, one runs *contoimpl* to produce *_impl.h* and *_impl.cc* files, as shown in the slide. (*Contoimpl* also produces an *_clientimpl.cc* file for any server-provided client-side code that may be needed for accessing an object, but this is rarely used.) In our example, *dbex_impl.h*, as produced by *contoimpl*, contains skeletal C++ declarations for implementation objects, i.e., the application objects being provided by servers to clients. These declarations contain everything except for the application code (which, of course, must be provided by the programmer). Similarly, *dbex_impl.cc* contains skeletal C++ definitions for the application code.

Client Side: Application Code

```
#include "dbex.h"
   // generated by contocc
#include <spring/util/
spring_lib.h>
#include <spring/naming/naming.h>

main( ) {
 dbex_NVcollection_fp db;
   // “fat pointer”: object reference

 try {
  db =
    naming<dbex_NVcollection>::
      resolve(domain_environment,
      "village/databases/capitals");
 } catch (...) {
  printf("can’t find it\n");
  exit(1);
 }

 db->add("RI", "Providence");
}
```

Explanation:

Assuming that some server is providing the NVcollection service for us, how do we access the service? Let’s further assume that the server has placed a reference to an NVcollection object in the name space (how it does so is discussed soon). We resolve the reference as shown in the slide. Since references to objects of arbitrary types may be bound to names and subsequently resolved, *resolve* is defined via a template. What is returned by *resolve* is a reference to our object (such references are known as “fat pointers”) and is assigned to *db*.

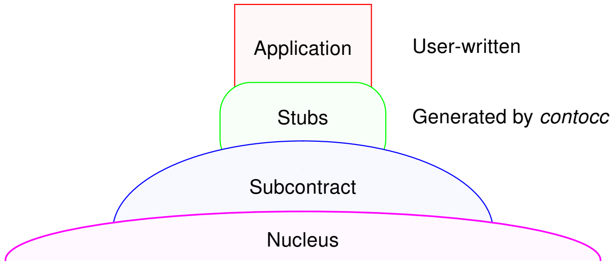
Note the type of *db*: it is a *dbex_NVcollection_fp*. This name was constructed by *contocc* by concatenating the module name with the interface name and then appending *_fp* to the end to indicate that this is an object reference (or fat pointer).

Once we obtain our object reference, we can invoke the methods of the object, just as if we had a pointer to a local object. (In fact, we might well have a pointer to a local object—Spring objects may be in the same address space as their clients or in different address spaces; there is no apparent difference.)

What really happens on the client side? Though we have the illusion of invoking a method of a local object, we are actually invoking a method of an object that is part of the stub code (supplied by the *contocc*-generated file, *dbex.h*). This stub code, through interactions with the code provided by the subcontract, in turn communicates with the server.

Using Spring Objects

Server Side

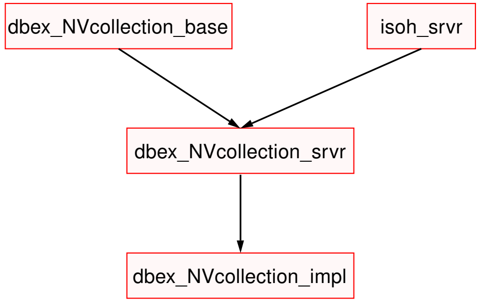


Explanation:

We now turn to the server side, which is layered as shown in the slide. This layering is realized partly through the use of inheritance: the server-side object implementation inherits (implementation-inherits) from code produced by *contocc*. This code is in turn linked to one or more subcontracts (one subcontract for the object being implemented, other subcontracts for objects being passed as arguments).

Using Spring Objects

Server Side: Class Hierarchy



Explanation:

Spring servers manage objects encapsulating not only the application but also the server-side stub code that handles communication with the client. In our example, *contocc* produces a skeleton for the *dbex_NVcollection_impl* object declaration, containing placeholders for the application writer to provide the application. This object inherits from *dbex_NVcollection_srvr*, which is completely provided by *contocc* and is, in essence, the server-side stub code. This, in turn, inherits from *dbex_NVcollection_base*, which is an abstract class that establishes the methods to be defined in *dbex_NVcollection_impl* as virtual methods, and from *isoh_srvr* ("invoke spring object handle"), which provides a number of standard methods important for the server-side stub.

Server Side: Implementation Class

```
// dbex_impl.h

#include "dbex_srvr.h"

class dbex_NVcollection_impl:
     public dbex_NVcollection_srvr {
public:
    dbex_NVcollection_impl( );//  added by programmer
    virtual string query(string name);
    virtual bool add(string name, string value);
    virtual bool remove(string name);
private:
    Node *search(string country, Node *node, Node **parent = 0);
    Node *head;
};
```

Explanation:

The slide shows the contents of the *contoimpl*-generated file, *dbex_impl.h*; the items in italics are provided by the application writer, the rest is provided by contoimpl. The *dbex_NVcollection_impl* class is the type of the application object, as perceived on the server side. The public methods generated by *contocc* are the methods made available to the client-side application, i.e., they are the methods mentioned in the IDL specification of the interface. To make this a usable C++ object on the server side, a number of other things must be added, in particular, a constructor and some private methods for use in the implementation of the object.

Note that there is no analog of the C++ notion of constructors within the Spring object model: clients cannot "will" an object into existence on the server, they can only obtain references to existing objects. However, as discussed starting on page 27, a client can "ask" that an object be created.

The application writer must of course provide definitions for the methods defined in the implementation object. *Contoimpl* provides a file containing a skeleton for these definitions. We do not show the completed *dbex_impl.cc* file in the slides, but a complete definition for the final form of our example appears in the source code accompanying this course.

Server Side: Object-Manager (Application) Code

```
#include "dbex_impl.h"
#include <spring/util/
spring_lib.h>
#include <spring/naming/naming.h>

main( ) {
 dbex_NVcollection_impl *db =
    new dbex_NVcollection_impl(
);
 try {
  naming<dbex_NVcollection>::
    rebind(domain_environment,
    "village/databases/capitals",
    dbex_NVcollection_fp(*db));
 } catch (...){
  printf("didn't work\n");
  exit(1);
 }

 springlib::become_server( );
}
```

Explanation:

Assuming that we have put together suitable definitions of the methods of *dbex_NVcollection_impl*, our next step is to produce the server that handles this class. As shown in the slide, this is easy. The server "wills" an implementation object into existence by invoking the constructor via a call to *new*. It then binds a reference to this object into the name space. It actually calls *rebind*, which works even if something is already bound with the given name (it replaces what was there previously). Many servers call *rebind* instead of *bind*, so that the binding succeeds even if the server neglected to *unbind* the reference the previous time it started up.

The server binds a reference to the object, not the object itself. A reference to a Spring object behaves like a pointer but is not really a pointer. Thus we cannot simply supply the address of the object, but must produce an official object reference (or "fat pointer"— see page 22). This is accomplished by using the operator *dbex_NVcollection_fp*, which is a member of the *dbex_NVcollection_srvr* class and is produced by *contocc*.

Once the server has published the reference to the object, it has nothing further to do but to handle requests for the object. These are in fact dealt with by other threads (set up by the runtime package), so the mainline thread, its duties completed, effectively retires (without terminating the entire process) by invoking *springlib::become_server*.

Thus we have completed the steps outlined on page 17.

Creating Objects

- ❏ Client wants to create an object
  - – call a *constructor*
  - – requires an existing object!
- ❏ Server provides special *factory* object
  - – supplies a *create* method for creating new objects

Explanation:

In C++ one creates an object by somehow invoking a constructor for it. But this notion does not work for Spring objects: Spring objects are managed by servers (which could be local but, in general, are remote). To cause an action on a server, a client must use a reference to an object handled by that server. Thus if there is no existing object reference, the client cannot cause actions to happen on the server; in particular, it cannot create an object on the server.

The solution to this dilemma is straightforward: supply the client with a reference to a special object via which the client can request that other objects be created. These special objects are known as *factory objects*.

Updated Contract

```
// dbex.idl

module dbex {
    interface NVcollection {
      string query(copy string name);
      boolean add(copy string name,
                  copy string value);
      boolean remove(copy string name);
    };

    interface NVcollection_factory {
      NVcollection create( );
    }
};
```

Explanation:

We add another interface to the *dbex* module, this one specifying a factory object. Our plan is that the server make available to clients a reference to the factory object; then clients, having resolved the factory object, can use it to create new instances of the *dbex_NVcollection* object.

Updated Environment Specification

```
// dbex.env

$singleton: sc=singleton
            sc_file="spring/subcontracts/singleton.h";
dbex::NVcollection:  $singleton;
dbex::NVcollection_factory:  $singleton;
```

Explanation:

If it is necessary to specify the subcontract explicitly, here is how to do it for our module that now has two interfaces.

Client Code with Factory

```
// Don't forget the header files ...

main( ) {
    dbex_NVcollection_factory_fp dbf;

    dbf = naming<dbex_NVcollection_factory>::
         resolve(domain_environment,
         "village/databases/factory");

    dbex_NVcollection_fp db = dbf->create( );
    db->add("Kansas", "Topeka");

    ...
}
```

Explanation:

Here is an example of how a client can utilize a factory to create, then make use of an object.

Addition to *dbex_impl.h*

```
class dbex_NVcollection_factory_impl:
     public dbex_NVcollection_factory_srvr {
public:
   virtual dbex_NVcollection_fp create( );
}
```

Explanation:

Given the new *dbex.idl* file, *contocc* produces a new *dbex_impl.h* file that contains the addition shown in the slide.

Updated Object Manager

```
main( ) {
   dbex_NVcollection_factory_fp dbf =
      dbex_NVcollection_factory_fp(
      *new dbex_NVcollection_factory_impl( ) );

   naming<dbex_NVcollection_factory>::
      rebind(domain_environment,
      "village/databases/factory",
      dbf);

   springlib::become_server( );
}
```

Explanation:

The object manager (i.e., server) is modified to bind a reference to a factory object in the name space.

The Factory

```
dbex_NVcollection_fp
      dbex_NVcollection_factory_impl::create( )
{
    dbex_NVcollection_fp fp;
    fp = dbex_NVcollection_fp(
       *new dbex_NVcollection_impl( ));
    return(fp);
}
```

Explanation:

The code for the create method of the factory is quite simple: it merely creates an instance of a *dbex_NVcollection_impl*, converts it into an object reference, and returns the reference.

Improvements

- ❏ New functionality
  - add a method that allows one to iterate through the contents of the collection
- ❏ New techniques
  - inheritance
  - communication of objects

Explanation:

It's now time to improve the service that we have been calling *dbex*. We add an *iterator* method that allows us to iterate through the contents of the NVcollection alphabetically. In making this improvement, we demonstrate two techniques: inheritance and communication of objects.

A New Contract

```
// newdbex.idl

import dbex;

module newdbex {
 struct NVpair {
  string name;
  string value;
 };

 interface collIter {
  NVpair next( );
 };

interface betterCollection :
    dbex::NVcollection {
   collIter getIter( );
  };

  interface
betterCollection_factory {
   betterCollection create( );
  };
}
```

Explanation:

The specification of our new service is shown in the slide. The iterator method, *collIter*, returns name/value pairs, which we declare as being items of type *NVpair*. A *struct* declaration behaves as an *interface* declaration; the difference is that struct objects are serverless—they exist locally and are communicated by simple copying.

We specify the *betterCollection* interface as inheriting from the *dbex::NVcollection* interface. This, of course, is interface inheritance: the actual implementation of a *betterCollection* can differ radically from that of *NVcollection* (though, as it turns out, we use implementation inheritance in this example). The *betterCollection* interface supplies a new method, *getIter*, which returns an iterator (of type *collIter*) for iterating through the given collection. So that *contocc* knows where to find the specification of the *dbex::NVcollection* interface, there is an *import* statement that indicates the file containing the specification.

Subcontracts ...

```
// newdbex.env

$singleton: sc=singleton
            sc_file="spring/subcontracts/singleton.h";
dbex: file = "dbex.idl"
dbex::NVcollection:  $singleton;
dbex::NVcollection_factory:  $singleton;
newdbex::NVpair:  $singleton;
newdbex::betterCollection:  $singleton;
newdbex::betterCollection_factory:  $singleton;
newdbex::collIter:  $singleton;
```

Explanation:

Again, just in case we need it, here is the environment file for the new example.

Improved Client Code (1)

```
// Don't forget the header files ...

main( ) {
    newdbex_betterCollection_factory_fp dbf;

    dbf = naming<newdbex_betterCollection_factory>::
        resolve(domain_environment, "village/databases/factory");

    newdbex_betterCollection_fp db = dbf->create( );

    db->add("RI", "Providence");
    db->add("Kansas", "Topeka");
    ...
```

Explanation:

A client of the new service resolves the factory, creates an instance of a *newdbex_betterCollection*, and puts some items into it.

Improved Client Code (2)

```
    newdbex_collIter_fp collIter = db->getIter( );
    newdbex_NVpair p = iter->next( );

    for (; p.name != ""; p = iter->next( )) {
     printf("%s : %s\n", (p.name).s2c( ), (p.value).s2c( ));
    }
}
```

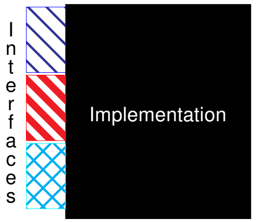
Explanation:

The client now obtains an iterator and uses it to list the contents of the collection in alphabetical order.

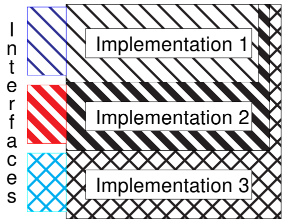
The code for the server is left for the exercises.

Inheritance

Interface Inheritance vs. Implementation Inheritance



Interface Inheritance



Implementation Inheritance

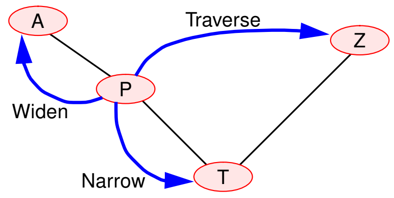
Explanation:

In the usual notion of implementation inheritance, the implementation of a subclass contains the implementations of the base classes. With interface inheritance, however, all that is required is that the subclass provide an implementation for all of its interfaces. For example, the *context* class requires an interface that includes *bind* and *resolve*. The *fs_context* class, which is derived from *context*, adds a new method, *create_file* (*fs_context* is used in file systems for directories). If we have a routine that expects to operate on a reference to a *context*, we can safely give it a reference to an *fs_context*, even though the implementations of the two have nothing in common other than the *context* interface. Thus, stating that *fs_context* inherits from *context* means not only that objects of type *fs_context* respond to all the methods to which objects of type *context* respond, but also that objects of type *fs_context* can be used wherever objects of type *context* are used.

We say that an instance of a subclass can be *widened* to be an instance of an ancestor class. Turning this around, an object that purports to be an instance of a class may also be an instance of a subclass. If it is, we can *narrow* the object so as to treat it as a member of the subclass.

Inheritance (2)

Fun With Inheritance



Explanation:

This figure depicts narrowing and widening: we have been given a reference to an object whose type we *perceive* to be *P*. Since the object at least can be treated as being of type *P* and we know that *P* is derived from *A*, we can always *widen* the object to *A*. This requires no runtime type checking, since *P* is a subclass of *A*.

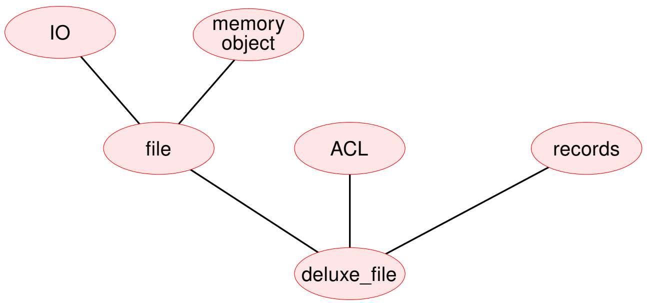
We may have received the reference to the object of type *P* as the result of invoking a method that was constrained to return a reference to *P* to us. However, the sender of the object may have had to widen an object of type *T*, derived from *P*, in order to meet the constraints. Thus we *perceive* the object to be of type *P* though its *true type* is *T*. In this case we can safely *narrow* our object and treat it as type *T*. Whether or not we can narrow an object depends upon information known only at runtime—some form of dynamic type checking is required.

Taking this a step further, if our object is really of type *T*, then (according to the picture), since *T* is also derived from *Z*, we should be able to treat our object as of type *Z* as well. This sort of transformation, called a *traverse*, clearly also requires the use of dynamic type checking.

We can perform widen, narrow, and traverse operations, where meaningful, on any Spring object. If dynamic type checking determines that the operation cannot be done, an exception is thrown.

Inheritance (3)

Example



Explanation:

Here is an example of the use of narrowing, widening, and traversing. Suppose that the usual sort of file supported by our system is of type *file*, which is derived from *IO* and *memory_object*. Thus, via widening, we can pass to any method that expects objects of type *IO* or *memory_object*. Suppose we have decided to support, in addition to *file*, a more sophisticated sort of file of type *deluxe_file*. This new type is derived from not only *file* but also *ACL* and *records*.

We might call a routine to obtain a reference to a file for us. This routine was written before *deluxe_file* was invented, so all it knows about is *file*. However, on obtaining the file, we can traverse the reference to be an *ACL* so that we can perform ACL operations on it, and then we can traverse the reference to be of type *records* so that we can perform record operations on it. Without such dynamic type manipulation, if the object we are given is said to be of type *file*, we could not safely coerce it to be a *deluxe_file*, an *ACL*, or a *records*.

Programmierung von Systemkomponenten

Beispiel: Nameserver

- ❑ Aufgabenstellung: Programmierung eines eigenen — zu den Systemschnittstellen konformen — Nameservers
- ❑ Vorgehensweise:
  - erforderliche Schnittstellendefinitionen suchen
    - Programmers Guide: The Name Service
    - /sys/interfaces/spring/naming
  - eigene Schnittstelle (Subtyp) beschreiben
  - Implementierung vornehmen

Programmierbeispiel: Nameserver

IDL-Beschreibung (ctx.idl)

```
import naming_context;
import name;

module ctx{
        interface bind_iterator
                : naming_context::binding_iterator {
        };
        interface ctx
                : naming_context::naming_context {
        };
};
```

- ❏ Schnittstellen werden unverändert übernommen!

Programmierbeispiel: Nameserver (2)

Subcontract-Konfiguration (ctx.env)

```
import "spring/spring.env";

ctx: file = "ctx.idl";
```

- ❏ Weitere Angaben sind hier nicht nötig, da die default-Einstellung (*singleton*-Subcontract) benutzt werden soll

Programmierbeispiel: Nameserver (3/4)

Makefile

```
# Regeln für typemgr- und contocc-Aufruf
include ContoccRules
# allgemeine Regeln zur Parametrierung von Spring-Tools
include /sys/etc/SpringRules

ENV = ctx.env

IDL_FILES = ctx.idl

# C++-Dateien für Server
SERVER_CFILES =        $(IDL_FILES:%.idl=%_srvr.cc) \
                       ctx_impl.cc csrvr.cc
SERVER_OFILES = $(SERVER_CFILES:%.cc=%.o)

# C++-Dateien für Client und Server
SHARED_CFILES =        $(IDL_FILES:%.idl=%.cc) \
                       $(IDL_FILES:%.idl=types_of_%.cc)
SHARED_OFILES =        $(SHARED_CFILES:%.cc=%.o)


PROGS = csrvr

all:    $(PROGS) $(SERVER_CFILES) $(SHARED_CFILES)

# Generierung des Servers (.o-Dateien werden nach Standardregeln aus
# /sys/etc/SpringRules und ContoccRules erzeugt
csrvr: ctx.o types_of_ctx.o ctx_impl.o ctx_srvr.o csrvr.o
        $(SDLD) -o csrvr ctx_impl.o ctx.o types_of_ctx.o \
                       ctx_srvr.o csrvr.o $(LIBRARIES)
# Generierung des Server-Skeletts
ctx_impl.cc: $(IDL_FILES) $(ENV)
        $(CONTOIMPL) -I/sys/interfaces -e $(ENV) $(IDL_FILES)
```


Programmierbeispiel: Nameserver (5/6)

Server-Programm

```
#include "ctx.h"
#include "ctx_impl.h"
#include <spring/util/spring_lib.h>
#include <spring/streams/stream.h>
#include <spring/naming/name.h>
#include <spring/naming/name_lib.h>
#include <spring/naming/naming.h>
#include <spring/naming/naming_lib.h>

int main(int argc, char **argv) {
  // Namensobjekt aus Zeichenkette generieren
  name_fp n_fp =
        name_lib::chars_to_name("machine/services/ctx");

  // Serverobjekt instantiieren und fat pointer erzeugen
  ctx_impl *ctx = new ctx_impl(n_fp);
  ctx_fp ctx_fp = ctx->make_fp();

  // Access-Liste für Serverobjekt erzeugen
  naming_context_access_list *nc_acl =
        new naming_context_access_list();
  // Eintrag in Access-Liste vornehmen
  acl_entry nc_acl_e("world", access_all, 0);
  nc_acl->addu(nc_acl_e);

  // Context-Objekt mit Namen n_fp in Namensraum binden, dabei explizit
  // angeben, daß es sich um einen Context handelt
  try {
    naming<naming_context>::rebind(domain_environment,
                              n_fp, ctx_fp,
                              naming_context_binding_type_ctx,
                              nc_acl);
  }catch(...) {
    cout << "naming<ctx>::rebind() failed !\n";
    exit(0);
  }
  // auf ankommende Methodenaufrufe warten
  springlib::become_server();
}
```

Programmierbeispiel: Nameserver (7)

Make-Aufruf

```
% make
    Generating stubs files from ctx.idl
/tools/contract/3.11.2/bin/contocc -fclient -fserver -f...
-e ctx.env ctx.idl

CC  -I/sys/interfaces -I/sys/interfaces/include -g   -c ctx.cc

/bin/typemgr -o types_of_ctx.tm -fno-database \
        -r /sys/lib/libspring.tm -i ctx.tm
sed 's/^const/extern const/' <types_of_ctx.cc > __tmp.cc
mv __tmp.cc types_of_ctx.cc

CC  -I/sys/interfaces -I/sys/interfaces/include -g   -c types_of_ctx.cc

contoimpl -I/sys/interfaces -e ctx.env ctx.idl

/tools/SC4.0-spring/bin/CC  -I/sys/interfaces -I/sys/interfaces/
include -g   -c ctx_impl.cc
"ctx_impl.h", line 15: Error: Type name expected instead of
"INSERT".
...
```


Programmierbeispiel: Nameserver (8/9)

ctx_impl.h

```
// ctx_impl.h generated by contoimpl (3.11.2) from ctx.idl
...
#include "ctx_srvr.h"

class ctx_bind_iterator_impl:
    public ctx_bind_iterator_srvr
{
public:
    // INSERT CONSTRUCTORS HERE;
    ctx_bind_iterator_impl(naming_context_binding_list &src_bl);
    ~ctx_bind_iterator_impl();
    // ---
   virtual void next(int arg_how_many, naming_context_binding_list
*&arg_bindings, bool &arg_more) throw ();
    // OTHER REDEFINED METHODS
    void _unreferenced();
private:
    // INSERT DATA MEMBERS AND PRIVATE METHODS HERE;
    naming_context_binding_list *nc_bl;
    int index;
};


class ctx_impl:
    public ctx_srvr
{
public:
    // INSERT CONSTRUCTORS HERE;
    ctx_impl(name_fp n_fp);
    // ---
    ...
    virtual named_fp resolve(name_fp arg_n, modes arg_m) throw ...;

    virtual void bind(name_fp arg_n, named_fp arg_obj,
naming_context_binding_type arg_m, naming_context_access_list
*arg_acl) throw ...;

    virtual void rebind(name_fp arg_n, named_fp arg_obj,
naming_context_binding_type arg_m, naming_context_access_list
*arg_acl) throw ...;

    virtual void unbind(name_fp arg_n) throw ...;
```

Programmierbeispiel: Nameserver (10/11)

ctx_impl.h (Fortsetzung)

```
    virtual naming_context_binding_info get_binding(name_fp arg_n,
bool arg_return_object, modes arg_m, bool arg_follow_symlink)
throw ...;

    virtual naming_context_binding_iterator_fp
get_all_bindings(name_fp arg_n, bool arg_return_objects) throw
...;

   virtual naming_context_stats stat() throw ();
...
private:
    // INSERT DATA MEMBERS AND PRIVATE METHODS HERE;
    naming_context_binding_info *get_info_by_name(name_fp n_fp);

    int own_create_time;
    naming_context_binding_list nc_bl;
    name_fp own_name;
};
```

ctx_impl.cc (Auszug)

```
// ctx_impl.cc generated by contoimpl (3.11.2)
// from ctx.idl
// Tue Jul  4 14:10:26 1995

#include "ctx_impl.h"

...
void ctx_impl::bind(name_fp arg_n, named_fp arg_obj,
naming_context_binding_type arg_m, naming_context_access_list
*arg_acl) throw ...
    // interface ::naming_context::naming_context
    // operation void bind(consume name::name n, consume
named::named obj, copy binding_type m, consume access_list acl)
{
  if (get_info_by_name(arg_n))
    throw naming_context_already_bound(arg_n);

  if (arg_m != naming_context_binding_type_multiple)
    throw naming_context_failure(
          naming_context_error_invalid_binding_type, own_name);
```


Programmierbeispiel: Nameserver (12/13)

ctx_impl.cc (Auszug)

```
  if (arg_n->length() != 1)
    throw naming_context_failure(
          naming_context_error_invalid_binding_type, own_name);

  naming_context_binding_info bi;
  bi.the_name = arg_n->get_comp(0);

  struct timeval tv;
  gettimeofday(&tv, 0);
  bi.create_time = tv.tv_sec;

  bi.bind_type = naming_context_binding_type_multiple;
  bi.obj = arg_obj;
  bi.acl = arg_acl;

  nc_bl.addu(bi);

  cout << "Binding-info added !\n";
}


naming_context_binding_iterator_fp
ctx_impl::get_all_bindings(name_fp arg_n, bool arg_return_objects)
throw (naming_context_not_found, naming_context_cannot_proceed,
naming_context_cannot_proceed_symlink, naming_context_failure,
naming_context_access_denied)
    // interface ::naming_context::naming_context
    // operation binding_iterator get_all_bindings(consume
name::name n, copy boolean return_objects)
{
  return naming_context_binding_iterator_fp(
            *new ctx_bind_iterator_impl(nc_bl)
         );
}
```