Spring Programming

2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No.: 8xx-xxxx-xx
Revision 0.1, April 1994

Please
Recycle

Contents

Preface

This document provides an introduction to Spring programming. It features a number of sample programs, which are available online, and can thus be inspected, compiled, and run. Their range is intended to cover the most fundamental aspects of Spring programming.

By concentrating on the practical essentials of program creation, this document encourages the reader to perform Spring programming experiments at the earliest opportunity. Often, therefore, it omits detailed feature analysis, and relies for its usefulness on the reader's initiative and willingness to experiment.

Prerequisites for Spring Programming

Before attempting the programming examples in this document, you need the following:

- Residence of MTV Building 19 – this is the only Sun building within which you can currently access Spring or the *Spring On Unix Prototype* (*SOUP*).
- Both of the following:
  - A UNIX host machine, established on the appropriate network, and running Solaris 1.x. This machine runs the console of your Spring machine, which you access with a *tip* connection.
  - An additional machine, which can be be designated for running the Spring Operating System. This should be a SPARCstation 2 or 10, with a minimum of 64 megabytes of memory, and a minimum of 8 megabytes of

available disk space (where the minimum files needed to run SpringOS will reside). Like the host machine, the designated Spring machine should be established on the network and should initially be running Solaris 1.x.

- Knowledge of UNIX and C++.
- Access to *~tjh/spring_progs*, which is the location of the described code examples.

How This Book Is Organized

Chapter 1, "Spring ProgrammingOverview," provides a brief, general overview of distributed object programming and the Spring operating system.

Chapter 2, "A Single Domain Program," presents and describes an example of Spring programming within a single *domain* or *address-space*. This chapter also gives a fundamental introduction to basic concepts and terms used in Spring programming.

Chapter 3, "Multiple Domain Programs," provides examples of distributed Spring programming, which involves passing arguments and returned values across two or more address spaces.

Chapter 4, "Interface Inheritance," describes the interface inheritance model, and demonstrates how it allows object substitutability.

Chapter 5, "Essential Spring Features," describes miscellaneous essential features that Spring provides to programmers. These include elements of the Contract interface definition language, such as exceptions, enums, structures, unions, and sequences (arrays). They also include the Spring C++ interfaces for strings, clocks, random number generators, tables, queues, and threads. The chapter also provides a description of Spring parameter passing modes, and of the equivalencies between primitive Contract types and C++ types.

Chapter 6, "The Name Service," discusses and exemplifies how objects in Spring can be bound to names, and accordingly resolved.

Spring Programming Overview 1

This chapter provides a general overview of the Spring Operating System, and of Spring programming, and introduces some of the terms and concepts that will be used frequently throughout this book.

1.1 The Spring Operating System

As a research project, the Spring Operating System was initiated with the intention of answering at least two significant questions, which had been perceived, through industry experience, as generally applicable to the design of distributed systems:

- How can such systems be made more capable of evolution and extension?
- How can networked programming be made less difficult?

Spring was immediately envisioned, therefore, as an operating system that would be sufficiently modular to allow frequent component-change, without incurring instability. It was also intended to provide a new level of support for the writing of distributed system services and applications.

From the project's inception, particular attention was given to the relationship within software systems between what were termed *interfaces* and *implementations*. An *interface* might be regarded as a formal declaration of the services offered by a software system for the use either of other systems or of itself; as such, the interface specifies the precise communication requirements that must be observed for these services to be activated. Accordingly, an

implementation is the code that provides services, in accordance with the service-definitions and associated communication-requirements specified by an interface.

As plans for Spring progressed, a number of fundamental design decisions were made with reference to these components:

- Throughout the system, there should be clear separation of interfaces from their associated implementations. Each interface should be regarded as a high-level definition of the services provided by its various implementations—it should not itself be formed as the result of any given implementation, since it would then be liable to invalidation by future implementation change. Instead, its implementations should be regarded as replaceable components, able to change in detail, without affecting the validity of the interface's specification.
- The interface technology should not restrict software components to given address spaces or machine boundaries. Instead, software components should be capable of distribution across address space and machine boundaries, with purely local procedure calls used for distributed process communication.
- The interface technology should be capable of evolution by means of an inheritance mechanism, so that new functionality might be added to old interfaces, without disturbing existing applications.

These decisions were felt to ensure the modularity and distributability of implementation code, while allowing interfaces—the fundamental service-definitions within the system—to be modified safely over time. And, this itself would allow the creation of a truly extensible, distributed operating system.

Proceeding from these basic design decisions, the *Spring Object Model* was defined; this would form the basis for all programs that might be part of the base Spring system, or might be added as services or applications by third parties.

1.2 *The Spring Object Model—An Introduction*

Spring programming is intended to facilitate the creation of programs that have strongly typed and well-defined interfaces, are easily extensible, and are *distributed*—that is, capable of interacting transparently across address spaces

and across machine boundaries. To achieve these goals, the system uses an object-oriented design that can be considered to exist on three levels: *interface*, *implementation*, and *subcontract*.

A Spring *interface* is a formal declaration of one or more operations. The declaration is written in *Contract*, which is a marginal superset of *IDL* (Interface Definition Language). An interface specifies only its operations' signatures—that is, the operation name, the return type, and the parameter types (if any). (Note that the terms *Contract* and *IDL* are used interchangeably in this document, though *IDL* is preferred.)

Each interface provides the basic specification for the design of a C++ object, to be created by a user as part of an *implementation* program. Such a program is typically referred to as a *server*, since it makes the interface-specified operations available as services to other C++ programs. The server's implementation code can take any form, provided that its objects and method-signatures correspond to those defined in the interface.

A C++ program that is intended to *use* (rather than provide) the operations defined by an interface is typically referred to as a *client*. The client accesses the objects that have been implemented by a server, and invokes the objects' methods.

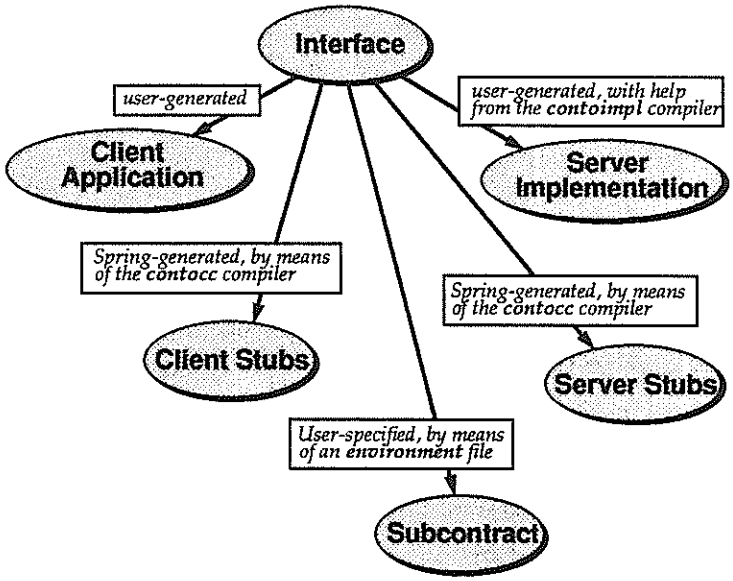
Importantly, server and client are written according to the strict rule that neither should know details of the other's implementation. The server only knows that the services it implements may be used by an unspecified client at some time. The client only knows that certain objects are available, and that they contain invokable methods. The precise means whereby this information is shared between server and client is regarded as a policy issue that might be resolved in various ways.

While Spring programming allows the user to work entirely in C++, it also underpins the implementation environment with *subcontract*, a technology that supports object distribution by providing method invocation and argument transfer across address-spaces. Thus, whenever a client invokes the methods of a C++ object that has been implemented from a Contract interface, execution appears to take place entirely within the client's own locale. In fact, the object's server may be in another address space, or on another machine on the network; the client remains unaware of the true locale, and the object's *subcontract* mechanism invisibly handles all issues related to communication between client and server.

The C++ implementation environment for both servers and clients is connected to the subcontract layer by means of Spring-generated C++ *stubs* code.

1.2.1 Spring Programming Components

The following diagram shows the required derivation of implementation, stubs, and subcontract components from a given interface:



As this diagram indicates, the *contoimpl* compiler can be used to provide skeletal C++ server implementation code based on a specified Contract interface: the user then completes the code by adding the appropriate implementation details.

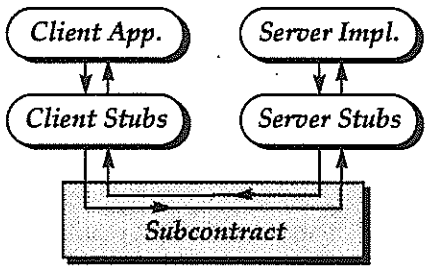
The *contocc* compiler generates the C++ code that links the implementation with the subcontract layer; some of this code in the form of **.h* files that are accessed by servers and clients through *include* statements.

The subcontract layer itself is specified with an *environment* file, which associates a given form of subcontract with each interface in a given Contract file. (The names of the Contract and environment files are specified as arguments when the *contoimpl* and *contocc* commands are executed.)

Once these components have been generated, the code can be fully compiled and run.

1.2.2 *Spring Client-Server Transactions*

The following diagram indicates the communication path between client and server programs:



When a client invokes the method of a Contract-defined object, the client-side stubs call the object's subcontract (whose identity has been specified in the appropriate *environment* file) and marshal the method-arguments into a buffer. The subcontract then transfers invocation to the server-side stubs: these duly unmarshal the arguments from their buffer, and invoke the server's implementation code, where method-execution occurs.

A result can be returned to the client in the same way, when server processing has completed.

1.2.3 *Spring Objects*

Each Contract interface that is implemented as a C++ class by a server must be instantiated as a *Spring object*, in order to be used in distributed programming. To effect this, a call to the constructor of the C++ implementation class is itself made the argument to a Spring-defined conversion operation, which returns a *fat pointer*—this is a Spring-defined type always used within C++ code to embody a Spring object-reference. (The terms *fat pointer* and *object reference* are used interchangeably in this document, though *object reference* is preferred.)

A fat pointer is itself a C++ class definition, which has as private members two C++ pointers: one to a *method table*, the other to a *representation*.

- The method table is comparable to a C++ virtual function table, but is language independent, and is constructed according to the special requirements of Spring distributed programming. The method table provides access to the server-implemented methods of the object defined by the interface.
- The representation varies in form, according to the type of the object: essentially, it is a target allocated at subcontract-level to the stubs of the interested client process, and allows method invocation to occur.

Whenever a client accesses an available object, it receives the fat pointer that corresponds to the object. The name of the fat pointer takes the form *objectname_fp*. The fat pointer can be treated as a regular C++ object-instantiation, though the methods it provides are to be accessed with the "->", rather than the " ." operator.

Note that use of the fat pointer is the closest that most users ever need to come to dealing with the mechanics of stubs and subcontract activity. Such activity is designed to happen automatically, and be hidden from the programmer of either server-implementations or client-applications.

1.2.4 *The Spring Name Service*

The Spring *name service* allows each object in the system to be associated with a name. To effect this, the service uses *contexts*, which are objects that contain defined name-to-object associations, known as *name bindings*. Thus, contexts are roughly analogous to directories in UNIX , while the names within any given context are analogous to files or subdirectories.

A *compound name* is an object-name that contains several components, indicating the object's relation to a context within which it is nested at some level: the compound name thus may have the appearance of a UNIX pathname. A *simple name* is a name with a single component. The total naming area spanned by a given context is known as a *name space*.

Although comparisons with the UNIX filesystem are useful, it is important to note that the Spring name service is far more inclusive. Whereas the UNIX filesystem does indeed only provide naming for files—the naming for other entities, such as processes or environment variables, being handled differently—the Spring name service provides a single model for the naming of *all* objects, whatever they represent.

Frequently, when a server process implements an object that it wishes to make available for the use of unspecified clients, it registers the object with the name service, specifying a name for the object, and the context in which it wishes the object to reside: this is known as *binding*. The choice of name is entirely arbitrary, and does not need to correspond to the name of the object's interface. When a client wishes to access a published object, it must reference the appropriate context and object name: this is known as *resolving*.

1.2.5 *Object Types and Inheritance*

In the Contract language, each interface is considered to define a *type*, much as a class is considered to define a type in C++. Thus, interface *foo* defines type *foo*.

Contract supports multiple inheritance of interfaces, and thus of types. When interface *foo* inherits from interface *bar*, *foo* is termed the *derived* interface; *bar* is termed the *base* interface. Provided that the definition of *foo* has specifically referenced *bar* as an inherited interface, *bar* is termed a *direct* base interface of *foo*. Any interfaces that *bar* itself has inherited, and are not referenced in the definition of *foo*, are termed *indirect* base interfaces of *foo*.

An interface that is implemented as an object determines the *true type* of the object. The server that provides the implementation can implement all methods of whatever direct and indirect base interfaces are inherited by the *true type* interface.

Though Spring programming principally uses a system of *static typing*, and expects that all possible type checks will be performed at compile time, it nevertheless also supports a certain amount of *dyanmic typing*. This allows base objects to be treated as derived objects (so called *narrowing*), and derived objects as base (*widening*). Additionally, one base object can be treated as another (*traversing*), provided that both of these base objects are indeed inherited by a common more derived object—no direct or indirect inheritance relationship need exist between the base objects themselves.

This *substitutability* of objects allows considerable freedom in the way objects are passed, bound, and resolved within Spring programs.

1.2.6 Spring Domains

The name *domain*, used throughout this document, signifies what might loosely be called a Spring *process*. However, a domain is properly defined as *an address space containing a set of threads*. Thus, a domain can actually have multiple Spring processes running within it—one to each thread.

1.2.7 Spring Features

Spring makes available a wide variety of programming features. Some of these are defined within the Contract language—exceptions, enums, structures, unions, and *sequences* (optimized arrays) can be defined in *idl* files much as interfaces, and can indeed be passed as arguments and returned as values.

Spring also provides pure C++ interfaces for strings, clocks, random number generators, hash tables, *queues* (linked lists of objects), and threads.

1.3 How this Document is Organized

This document consists of six chapters. The first of these is the current overview. Chapters two to four each present a fundamental aspect of Spring programming—each is based around one or more code examples, whose online locations are provided. Chapters five and six are respectively dedicated to *Essential Spring Features* and *The Name Service*.

The reader is recommended to work through the book, chapter by chapter, but in each case start by examining the online code, rather than the text of the document. The code should be inspected, as should the Makefile and the README file, which is present in each example-directory. The program should be made and then run. Having done this, the reader is free to examine the text of this document, and so find pertinent details on how the program is constructed and built.

However, if the online code is not available, the book can still be read through in the normal fashion. Most significant details of the code are reproduced in the book.

1.3.1 Overview of Online Code Examples

The online code examples described in this document are all contained in subdirectories located beneath the following location:

```
~tjh/spring_progs
```

Within each subdirectory are additional subdirectories, each of which contains the code for a single Spring program. Each directory also contains a README, which indicates briefly the contents of the directory, the elements that are of specific interest, and the command(s) required for running the program. Note that each successive program is designed as a variant on the previous one; thus, a sequence of more or less incremental change is established from the first program to the last.

Each directory that contains a program also contains a file named *VarRules*. This file itself contains environment variables that specify the location of the Spring file hierarchy from which headers and other files are obtained. If you elect not to run the programs in their current location, and instead copy them to a new location, you must modify this file. You may also need to modify the Makefiles, depending on other environmental changes you make. See the *SpringOS Process Handbook* for details on environment variables, the *VarRules* file, and other related issues.

The following is a listing of all Spring programs, with an indication of their content.

Note – From time to time, the code exerpted and reprinted *in this hardcopy document* might become out of date—while every attempt is made to keep it accurate, the comparative frequency of changes in the Spring development environment make it likely that temporary inconsistencies will occur. The *online* code examples are currently the top priority for update and evolution. Thus, please consult the online code for the best and most recent examples of what should be built and run.

- */home/tjh/spring_progs/chap2/foo*
  A program that runs in a single address space, performing a simple arithmetical exercise. Indicates the rudiments of how a Spring program is put together.

- */home/tjh/spring_progs/chap3/distrib_foo*
  A program that runs in two address-spaces. Indicates the rudiments of cross-address-space communication.
- */home/tjh/spring_progs/chap3/distrib_foo/lib_version*
  Same as the program immediately above, but additionally demonstrating use of client implementation libraries, which are provided by the server relieve the client of the need to know the server-determined context of the required object.
- */home/tjh/spring_progs/chap3/multi_distrib_foo*
  A program that runs in three address-spaces. Indicates how a program can function as a server and a client simultaneously.
- */home/tjh/spring_progs/chap3/multi_distrib_foo/lib_version*
  Same as the program immediately above, but additionally demonstrating use of client implementation libraries.
- */home/tjh/spring_progs/chap4/inherit*
  A program that demonstrates multiple inheritance of interfaces—three interfaces are inherited by a fourth. The program runs in three address-spaces.
- */home/tjh/spring_progs/chap4/inherit/subst/naming/example1*
  /home/tjh/spring_progs/chap4/inherit/subst/naming/example2
  /home/tjh/spring_progs/chap4/inherit/subst/naming/example3

  Programs that explore the possibilities of of object substitution that are raised by interface inheritance. These examples demonstrate different ways of name-binding and resolving objects, in terms of their base and derived types.
- */home/tjh/spring_progs/chap4/inherit/subst/passing/nrw_and_trvrse*
  Another program that demonstrates object substitution, this time by "narrowing" and "traversing", used in object passing and method invocation. The program runs in four address-spaces.
- */home/tjh/spring_progs/chap4/inherit/subst/passing/widen*
  Object substitution by means of "widening", used in object passing and method invocation. Again, the program runs in four address-spaces.

- */home/tjh/spring_progs/chap4/my_inherit*
  A program that demonstrates use of concrete types. A single derived type is made the basis for server implementation, all functionality being provided by its base types. The client programs against the base types.
- */home/tjh/spring_progs/chap5/clock*
  A program that demonstrates use of the Spring C++ clock object
- */home/tjh/spring_progs/chap5/enums*
  A demonstration of Spring *enums*, which can be specified in the Contract language, within an *idl* file.
- */home/tjh/spring_progs/chap5/exceptions*
  Use of exceptions, which allow the definition of routines to be executed in the event of an invocation causing an unexpected result. Exceptions are defined in the Contract language.
- */home/tjh/spring_progs/chap5/param_modes*
  A program that demonstrates Spring parameter passing modes.
- */home/tjh/spring_progs/chap5/queues*
  A demonstration of the Spring C++ objects called *queues*, which provide the ability to create objects in linked lists.
- */home/tjh/spring_progs/chap5/random*
  A demonstration of the Spring C++ interface *random.h*, which provides a random number generator.
- */home/tjh/spring_progs/chap5/sequences*
  Use of Spring *sequences*, which are unbounded arrays.
- */home/tjh/spring_progs/chap5/strings*
  A demonstration of the Spring C++ *string* object and its methods.
- */home/tjh/spring_progs/chap5/structures*
  A program that illustrates Contract-defined structures.
- */home/tjh/spring_progs/chap5/tables*
  A program that demonstrates use of Spring C++ hash table objects, and their defined methods.

- */home/tjh/spring_progs/chap5/threads/three_threads*
  /home/tjh/spring_progs/chap5/threads/main_thread
  Programs that illustrate use of threads, conditions, and mutexes, all of which are defined as Spring C++ objects.
- */home/tjh/spring_progs/chap5/unions*
  A demonstration of how to use Contract-defined unions.
- */home/tjh/spring_progs/chap6/context*
  How to use the Spring Name Service to bind, resolve, rebind, and perform additional operations on a naming context.
- */home/tjh/spring_progs/chap6/dup*
  A program that shows the part of the Spring Name Service that duplicates objects.
- */home/tjh/spring_progs/chap6/freeze*
  A demonstration of the freezing facility provided by the Spring Name Service—this is principally an internal function of the Name Service, and is used transparently in the binding and resolving of objects.
- */home/tjh/spring_progs/chap6/general*
  A program that demonstrates various forms of functionality associated with the Name Service, including retrieval of information on existing object bindings.
- */home/tjh/spring_progs/chap6/name*
  An illustration of Spring *name* and *name_component* objects—these are not intended for normal client-programmer use, but are discussed since they may at times need to be used.
- */home/tjh/spring_progs/chap6/om*
  A program that demonstrates ordered merges in Spring.
- */home/tjh/spring_progs/chap6/symlink*
  A demonstration of how to create a symbolic link.

A Single-Domain Program 2

This chapter is concerned with Spring programming within a single *domain* or *address-space*. Like other chapters of this document, it reproduces and discusses a code example, which is available online, and can be compiled and run. The chapter uses the code example to introduce and define various significant concepts and terms employed in Spring programming.

You are recommended to experiment with the code example before proceeding further. The code is in the following locations:

```
~tjh/spring_progs/chap2/foo
```

The directory contains a *README* file, which provides basic details of how the program is created. When you have read the *README* file, and have built and run the code, you can continue by reading this chapter.

2.1 Required Steps

To write and run a single-domain program, eight major steps are required. These are briefly summarized below.

1. **Interface Definition**
   This is accomplished by writing a *Contract* (or *IDL*) file, and requires the program designer's knowledge both of the nature of the services that the C++ server program will provide, and of the necessary appearance of the corresponding operations' signatures.

2. **Subcontract Specification**
   Subcontracts are the Spring services that provide distributed method invocation and argument marshalling for defined interfaces. They are specified by writing an *environment* file, which associates them with defined IDL interfaces.

3. **C++ Implementation of Defined Interfaces**
   This involves running the Spring *contoimpl* compiler to provide *skeletal implementations*—these are files containing C++ code templates, within which the desired implementation code can be keyed-in.

4. **Client Main Program Creation**
   The main program contains the appropriate client invocations on server-object methods.

5. **Translation of Interfaces from *Contract* to C++**
   This is accomplished by running the *contocc* compiler. This generates the C++ code for client- and server-side *stubs*, whose header files are included by clients and servers. (Note that the appropriate *includes* will already have been placed in the implementation files by *contoimpl*.)

6. **Definition of Type IDs for Interfaces**
   This is accomplished by running the Spring *typemgr* program.

7. **Compilation**
   This is performed with the Gnu *g++* compiler.

8. **Execution**

In this example, the single address-space program is created in a single directory. To emphasize the nature of the different steps of the creation process, necessary commands are initially demonstrated within this document as entered against the SpringOS prompt. The appropriate Makefile is shown later.

2.2 *Factories and Object Availability*

Before examining in detail the major steps required for single-domain program creation, the nature and relevancy of *factories* should be described.

Spring requires that client and server remain separate, knowing nothing of each other's implementation code or address-space location. Nevertheless, to enable a client to use the objects it provides, a server must share some piece of information with the client. The information should allow the client to access server-provided objects, and make invocations on them.

A *factory* is one means whereby control of object-creation can be handed to a client. A *factory* is a distributed object that provides a method for creating another object—the one that is actually desired by the client. The method may take parameters for passing initialization-arguments to the desired object's constructor. Thus, accessing the factory allows a client to create any number of desired objects, and instantiate each one differently.

Defined by the server implementation, the *factory* itself can be given to the client in the form of a **.o* or **.cc* file.

Note that use of a factory may not always be necessary: for example, when a client requires a single object of a given type, and there is no requirement that initializations be performed, a single object can be accessed by the client, and simply used in the form in which it is obtained.

2.3 *The Single-Domain Program—Analysis*

This section provides an analysis of the sample single-domain program, and explains how it is constructed and compiled.

2.3.1 *Interface Definition*

This example creates an interface that provides two operations on an integer value—one for setting, one for getting.

A second interface is required in order to create a *factory*, named *foo_factory*. The factory contains a single operation, *create*, which creates and returns *foo* objects. Thus, the client code that holds a *foo_factory* object is free to create any number of *foo* objects, with the integer value of each initialized as appropriate.

```
interface foo {
    long get_value();
    void set_value(copy long new_value);
};
```

```
interface foo_factory {
    foo create (copy long initial_value);
};
```

The *set_value* and *create* methods each take an integer parameter. This is specified as a *long*, which is the Contract library type used to specify the C++ type *int*. Each parameter is specified with the *copy* parameter passing mode, which is somewhat analogous to the *call by value* mode used in various programming languages.

The name of an interface definition file should be of the form *name.idl*. Thus, the current file is named *foo.idl*.

2.3.2 Subcontract Specification

To specify *subcontracts*, which are the Spring services that provide distributed method invocation and argument marshalling for defined interfaces, an *environment* file is written. This associates created interfaces with the appropriate subcontracts.

In this example, each line begins with a symbol representing the name of the Contract interface. An *sc=* attribute is used to specify the symbol's subcontract, which here, in both cases, is *singleton*, a standard subcontract for Spring objects. An *sc_file=* attribute specifies the location of the interface's subcontract class, which in this case is *singleton.h*.

```
foo: sc=singleton sc_file="subcontracts/singleton.h";
foo_factory: sc=singleton sc_file="subcontracts/singleton.h";
```

The name of an environment file should be of the form *name.env*. Thus, the current file is named *foo.env*.

2.3.3 C++ Implementation of Defined Interfaces

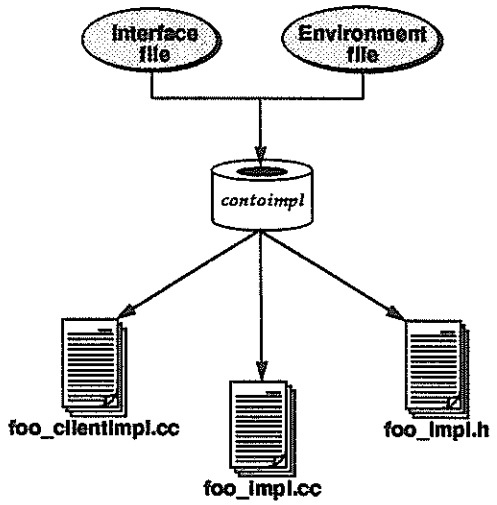
Server implementation consists of C++ class and method definitions that are analogue equivalents of the defined Contract interfaces. To assist in the creation of server implementations, Spring provides a program named *contoimpl* (that is, *Contract to Implementation*), which generates *skeletal* implementations—these are files containing partially written C++ code as a framework within which the user's desired implementation code can be specified.

Note – At the moment, *contoimpl* can be run only on UNIX. It is expected that the program will soon run on Spring.

The existing *.env* and *.idl* files are required as arguments to the *contoimpl* command:

```
/tools/contract/3.7.0/bin/contoimpl -e foo.env foo.idl
```

The following diagram indicates the effect of running the program:



Three new files are created by *contoimpl*:

- *foo_clientimpl.cc* is intended to contain a client-side implementation—server-provided code that allows a client to access and use the server's objects. This file might be used to provide a library for the client to link against.

  The exact nature of the client implementation code is not possible to predict; thus, the file is empty, except for comments indicating the file name and time of generation. In some implementations, the file may not be needed, and can in such cases be deleted.

- *foo_impl.h* contains skeletal C++ class definitions for the analogue classes based on the interfaces in the specified Contract file.

- *foo_impl.cc* contains skeletal method definitions for the analogue classes in *foo_impl.h*.

All three files generated for this example are described below.

foo_impl.h

foo_impl.h contains the C++ class definition for *foo* and *foo_factory*. When generated by *contoimpl*, its initial appearance is as follows:

```
// foo_impl.h generated by contoimpl (3.7.0)
// from foo.idl
// Wed Sep  1 12:47:55 1993

#pragma once
#pragma interface

#include "foo_srvr.h"

class foo_impl:
    public foo_srvr
{
public:
    INSERT CONSTRUCTORS HERE;
    virtual int get_value();
    virtual void set_value(int arg_new_value);
private:
    INSERT DATA MEMBERS AND PRIVATE METHODS HERE;
};

class foo_factory_impl:
    public foo_factory_srvr
{
public:
    INSERT CONSTRUCTORS HERE;
    virtual foo_fp create(int arg_initial_value);
private:
    INSERT DATA MEMBERS AND PRIVATE METHODS HERE;
};
```

As this example shows, the raw *contoimpl*-generated file contains indicators as to where implementation code should be typed. A completed version of the file is shown below:

```
// foo_impl.h generated by contoimpl (3.7.0)
// from foo.idl
// Wed Sep  1 12:47:55 1993

#pragma once
```

```
#pragma interface

#include "foo_srvr.h"

class foo_impl:
    public foo_srvr
{
public:
    foo_impl(int initial_value) : the_value(initial_value) {}
    virtual int get_value();
    virtual void set_value(int arg_new_value);
private:
    int the_value;
};

class foo_factory_impl:
    public foo_factory_srvr
{
public:
    virtual foo_fp create(int arg_initial_value);
};
```

A private data member has been added to *foo_impl*, as has a constructor method, which initializes the private data member. Note that a constructor needs to be added only if it will perform initializations or other activities. A default vanilla constructor— in the form *foo_impl() {};*—is automatically provided with each C++ implementation class, and does not need to be declared. (Indeed, if one is specified, a compilation error may occur.)

Note that the *contoimpl*-provided names of implementation classes—which take the form *interface_name_impl*—are optional, and can be changed manually.

The result of the *foo_factory_impl create* method is specified as *foo_fp*, rather than just *foo*. This is because the result of the implementation method will be a *fat pointer*, the Spring-defined C++ object-reference that connects a C++ implementation to the underlying Spring distributed object mechanisms. This is the standard form that a Spring object—whether of remote or local origin—takes in the local C++ environment of any client or server. The object reference is like a normal C++ class instantiation, though its methods are invoked with the -> operator (not the "." operator)—that is, precisely as if it were a pointer to an instantiation, rather than the instantiation itself.

The *pragma once* line protects header files against multiple inclusion. The *pragma interface* line is used by header files to minimize the production of redundant copies of inlines and C++ *vtbls*.

The file contains an *include* for *foo_srvr.h*, which is discussed below.

foo_impl.cc

The *foo_impl.cc* file is intended to contain the actual method definitions for *foo*. Once again, the template contains insertion indicators—this time as a single-line *INSERT CODE HERE* statement within the otherwise empty parameters of each method. A duly completed file appears as follows:

```
// foo_impl.cc generated by contoimpl (3.7.0)
// from foo.idl
// Wed Sep  1 12:47:55 1993

#include "foo_impl.h"

int foo_impl::get_value()
    // interface ::foo
    // operation long get_value()
{
    return this->the_value;
}

void foo_impl::set_value(int arg_new_value)
    // interface ::foo
    // operation void set_value(long new_value )
{
    this->the_value = arg_new_value;
}

foo_fp foo_factory_impl::create(int arg_initial_value)
    // interface ::foo_factory
    // operation foo create(long initial_value )
{
  foo_fp result = foo_fp(*new foo_impl(arg_initial_value) );
  return result;
}
```

The argument to the *set_value* method appears as *arg_new_value*, while the argument specified in the Contract file was simply *new_value*. The *contoimpl* program prepends *arg_* to every argument name that was specified within an interface. This should be remembered when arguments are referenced in implementation code.

The *create* method for *foo_factory_impl* manufactures a *foo* object reference by dereferencing a call to *new*, which is made on the constructor for *foo_impl*. This is the standard means of creating an object reference within the server implementation environment.

foo_clientimpl.cc

When created by *contoimpl*, the *foo_clientimpl.cc* file is blank, save for some commented header information. The file is intended to provide a means by the server can make available to the client the object references it will require. Thus, it might contain server-side *includes*, definitions, and calls that are necessary for this, but should be kept separate from the client itself, as should all true server-side implementation details.

```
#include "foo_impl.h"
#include "util/basic.h"

foo_factory_fp make_factory () {
  foo_factory_fp factory = foo_factory_fp(*new foo_factory_impl() );
   return factory;
}
```

This code defines a function that creates and returns a *foo_factory* object reference. This object reference will be made available to the client. The file includes *foo_impl.h*, which contains the class definition for *foo_factory*. It also includes *util/basic.h*, which contains basic Spring definitions that should be included by all **.cc* files, and specifically supports the creation of factories.

2.3.4 *Client Main Program Creation*

A sample *client.cc* main program is shown below:

```
#include <spring/streams/stream.h>
#include "foo.h"

foo_factory_fp make_factory();
```

```
void print (char *text, foo_fp f)
{
  cout << text << f->get_value() << "\n";
}

foo_fp add(foo_fp f1, foo_fp f2, foo_factory_fp ff)
{
  return ff->create(f1->get_value() + f2->get_value() );
}

int main()
{
  foo_factory_fp foofac = make_factory();

  foo_fp alpha;
  alpha = foofac->create(3);

  foo_fp beta;
  beta = foofac->create(7);

  print("alpha: ", alpha);
  print("beta: ", beta);

  foo_fp sum;
  sum = add(alpha, beta, foofac);

  print("sum: ", sum);

  return 0;
}
```

The program prototypes the *make_factory* funcstion from *foo_clientimpl.cc*, and uses this function to create a *foo_factory*, which is itself used to return *foo* object references. The program also defines two functions of its own: one to print integers to standard output, the other to return a *foo* object reference whose integer member is the sum of those of two other *foo* object references.

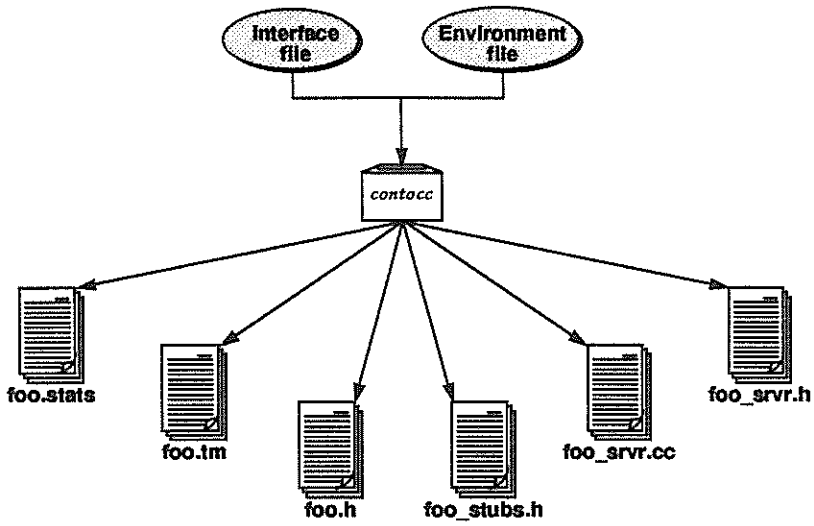
The file contains an *include* for *foo.h*, which is discussed below.

2.3.5 Translate Interfaces from Contract to C++

Spring provides a compiler named *contocc* (that is, *Contract to C++*), which translates interfaces described in a Contract (**.idl*) file to a corresponding set of C++ (**.cc* and **.h*) files. The C++ files are known as client- and server-side

stubs; their role is to connect the C++ implementation environment with Spring's underlying subcontract mechanisms. Object references, for example, are defined in the stubs.

The following diagram demonstrates use of *contocc*:



The files created are as follows:

- *foo.stats* contains information related to special use of Spring method tables. This file is not used in the current example, and will be discussed later in this document.
- *foo.tm* is a type database file, which contains information on compile- and run-time type definitions. The file contains a header, which contains information on the tool version and the time the file was created, followed by a series of records that describe Contract types. The *foo.tm* file will be used below by the *typemgr* program to create a **.cc* file, used in final compilation.

  See the *tm(5)* manpage for further information.
- *foo.h* contains the C++ stubs required by the client. Thus, this file should be included by all client code that wishes to make invocations on objects implemented by the server.

The file defines the class *foo_fp*, which is the object reference for the object. This contains a number of methods that allow basic manipulation of object references, such as ->, ==, and !=. It also contains as private data the pointers to the representation and method table for *foo*.

foo.h also defines the class *foo*, which is the object itself. Although this class is never employed directly by the user, it contains a number of client-accessible C++ method definitions, including those for subcontract-level copying, consuming, and argument marshalling, and provides client-side implementations of the object's interface-specified methods.

Finally, *foo.h* contains the class definition for the method table accessed by the class *foo_fp*.

- *foo_stubs.h* provides header code required by both client-side and server-side stubs.
- *foo_srvr.cc* and *foo_srvr.h* contain the C++ stubs required by the server. These include the definition of the class *foo_srvr*, which is inherited by the *foo_impl* class—as previously shown, *foo_srvr.h* is included in the *foo_impl.h* implementation file.

  Note that *foo_srvr.h*, though a server-side stubs file, nevertheless includes the client-side stubs file *foo.h*. This is because *foo.h* defines the object reference for the object, which the server needs in order to create and return object references when necessary. Thus, all three stubs files are actually required by the server, while only one, *foo.h*, is required by the client.

The following command runs the *contocc* compiler:

```
/bin/contocc_3_8_1 -fclient -fserver -e foo.env foo.idl
```

The files *foo.stats*, *foo.tm*, *foo.h*, *foo_srvr.h*, and *foo_stubs.h* now appear in the current working directory.

2.3.6 *Definition of Type IDs for Interfaces*

The *typemgr* program defines Spring type IDs for all interfaces within a given file. For single address-space code, the program might be executed with the following line:

```
/bin/typemgr -o types_of_foo.tm -fno-database -r \
/home/sarum/os/sys/lib/libspring.tm -i foo.tm
```

This command (which is fully described by the *typemgr* manpage) takes as its argument the *foo.tm* type database file that was created by *contocc*. It also references *libspring.tm*, which contains type definitions for globally used Spring objects. The command creates an additional file in the working directory, *types_of_foo.cc*, which provides the type ids in a form appropriate for inclusion in the final compile.

2.3.7 Compilation and Linking

The following lines compile and link the program for Spring:

```
/bin/gcc.1.94.9 -g -O -I/home/sarum/os/interfaces \
-I/home/sarum/os/interfaces -I/home/sarum/os/interfaces/spring \
-I/home/sarum/os/interaces foo.cc types_of_foo.cc\
foo_clientimpl.cc foo_srvr.cc foo_impl.cc client.cc
/bin/ld.2.1.92 -L/home/sarum/os/sys/lib \
-L/home/sarum/os/sys/lib -C -e start /home/sarum/os/sys/lib/crt0.o\
/home/sarum/os/sys/lib/drt0.o -o foo foo.o types_of_foo.o \
foo_srvr.o foo_impl.o foo_clientimpl.o client.o -lspring
```

2.3.8 Execution

To execute the program, you must be running the Spring Operating System. When Spring is running, the program can be executed, and provides the following output:

```
sarum% ./example
alpha: 3
beta: 7
sum: 10
```

2.3.9 Makefile Usage

The following Makefile performs all the steps in the above procedure that occur following implementation of interfaces and creation of the *main* program:

```
include VarRules
include $(OS)/Rules
include $(OS)/PT_Rules

ENV = foo.env
IDL_FILES = foo.idl
CFILES = $(IDL_FILES:%.idl=%_srvr.cc) $(IDL_FILES:%.idl=%.cc) \
        $(IDL_FILES:%.idl=types_of_%.cc)
OFILES = $(CFILES:%.cc=%.o) foo_impl.o foo_clientimpl.o client.o
```

```
PROGS = foo

$(OS)/%.h:

%.h + %.cc + \
%_stubs.h + \
%_srvr.h + %_srvr.cc + \
%.tm + %.stats: %.idl
    @echo "    Generating $@ from $<"
    $(CONTOCC) $(CONTOCC_FLAGS) $(INCLUDES) -e $(ENV) $*.idl
    @$(MAKE) -t "ENV=$(ENV)" \
         $*.h $*.cc $*_stubs.h $*_srvr.h $*_srvr.cc $*.tm $*.stats \
        > /dev/null

types_of_%.cc: %.tm
    $(TYPEMGR) -o types_of_$*.tm -fno-database \
        -r $(OS)/sys/lib/libspring.tm -i $*.tm

all: $(CFILES) foo_impl.cc foo_clientimpl.cc client.cc $(PROGS)

foo: $(OFILES)
     $(SDLD) -o foo $(OFILES) $(LIBRARIES)

clean:
    $(RM) $(PROGS) $(OFILES) *~ core

superclean:
    $(RM) $(PROGS) $(OFILES) *~ core $(CFILES) foo.h foo_srvr.h \
    foo_stubs.h *.stats *.tm
```

The Makefile contains include lines for three files: *VarRules*, *$(OS)Rules*, and *$(OS)/PT_Rules*. The appearance of *VarRules* is typically as follows:

```
# Edit this variable to the current directory path
DIR=/home/tjh/spring/foo/others/distributed/foo

# Edit this variable to point to the OS you use for headers and
installation
OS=/home/sarum/os

#Point this variable to the installation dir
INSTALL_DIR=${OS}
```

The comments in this file are self-explanatory—the other two included files, *$(OS)Rules* and *$(OS)/PT_Rules*, are located in the directory specified by the *OS* variable, and provide various definitions that are required for compilation. A complete description of these files and their roles is provided in the *SpringOS Process Handbook*.s

The Makefile also contains rules for the *contocc* and *typemgr* programs, specifying the **.cc*, **.h*, and other files that must be derived from the given **.idl* file. Note that the macros used in this Makefile are chiefly defined in the included *Rules* file, or in the files that it itself includes.

Multiple-Domain Programs 3

This chapter is concerned with Spring programming across multiple domains. Like the previous chapter, it reproduces and discusses code examples, which are available online, and can be compiled and run.

The code described in the first section of this chapter—Section 3.2—is in the following locations, which respectively contain the Spring- and SOUP-runnable versions:

```
~/tjh/spring_progs/chap3/distrib_foo
```

Once more, you are recommended to examine and experiment with the above code prior to reading further. Each directory contains an appropriate README and a Makefile. Note, again, that the code for both Spring and SOUP must be compiled on UNIX, due to the configuration chosen for the Makefiles.

3.1 A Two-Domain Program

This section modifies the program from Chapter 2 so that it runs as two separate processes, each in its own address space.

This example differs from the previous one in the following respects:

- It does not have a single *main* file—it has two, named *client*.cc and *server*.cc, which are compiled into the *client* and *server* executables respectively.
- The *foo_clientimpl.cc* file is not used. The client's access to the objects it requires is coded within the client's own main program.

- In consequence of address space separation, the server makes its objects remotely available; it uses a factory object and the *Spring name service*.

This section introduces several additional details of basic Spring programming, such as the nature and purpose of various commonly used *include* files.

3.1.1 Interface Definition

```
interface foo {
    long get_value();
    void set_value(copy long new_value);
};

interface foo_factory {
    foo create (copy long a);
};
```

The Contract file contains two interfaces. The *foo* interface is similar to that from the previous example.

A foo_factory, rather than *foo* itself, is made available to the client: this allows the client to call the *create* method, and thus manufacture multiple copies of the *foo* object.

3.1.2 Subcontract Specification

```
foo: sc=singleton sc_file="subcontracts/singleton.h";
foo_factory: sc=singleton sc_file="subcontracts/singleton.h";
```

3.1.3 C++ Implementation of Defined Interfaces

The *contoimpl* call [*which currently must be made on UNIX*] is the same as in the example in Chapter 2:

```
/tools/contract/3.7.0/bin/contoimpl -e foo.env foo.idl
```

3.1.3.1 foo_impl.h

```
// foo_impl.h generated by contoimpl (3.7.0)
// from foo.idl
// Mon Aug 16 10:36:30 1993

#pragma once
```

```
#pragma interface

#include "foo_srvr.h"

class foo_impl:
    public foo_srvr
{
public:
    virtual int get_value();
    virtual void set_value(int arg_new_value);
private:
    int the_value;
};

class foo_factory_impl:
    public foo_factory_srvr
{
public:
    virtual foo_fp create(int arg_a);
};
```

No constructor method is specified for either class; thus, a default constructor, performing no initializations, is provided for each.

3.1.3.2 *foo_impl.cc*

```
// foo_impl.cc generated by contoimpl (3.7.0)
// from foo.idl
// Mon Aug 16 10:36:30 1993

#include "foo_impl.h"

int foo_impl::get_value()
    // interface ::foo
    // operation long get_value()
{
    return this->the_value;
}

void foo_impl::set_value(int arg_new_value)
    // interface ::foo
    // operation void set_value(long new_value )
{
    this->the_value = arg_new_value;
}
```

```
foo_fp foo_factory_impl::create(int arg_a)
    // interface ::foo_factory
    // operation foo create(long a )
{
  foo_fp f;
  f = foo_fp (*new foo_impl() );
  f->set_value(arg_a);

  return f;
}
```

The *create* method defined by *foo_factory_impl* creates a *foo* object reference within the server address space. When the *foo_factory* object reference is provided via the naming service as a distributed service to the client, the client calls the *create* method to provide *foo* object references for its own local use. Spring's subcontract mechanism will cause the *create* method to be invoked within the server domain, and will pass the resulting object reference back to the client.

3.1.4 *Server Main Program Creation*

As described above, two *main* programs are needed, one to perform client, the other to perform server activities. The server program must make available to the client the objects it expects the client to require.

A server may use any appropriate means to make its object available. In many cases, use of the Spring *name service* is recommended. The server registers the object with the name service, specifying a name for the object, and the context in which it wishes the object to reside: this is known as *binding*. The choice of name is arbitrary, and does not need to correspond to the name of the object's interface. When a client wishes to access a published object, it must reference the appropriate context and name: this is known as *resolving*.

In the server main program shown below, the server binds an object of type *foo_factory* under the name *the_factory*:

```
#include "foo_impl.h"
#include "naming/naming.h"
#include "util/spring_lib.h"
#include <spring/streams/stream.h>

declare_naming(foo_factory);
```

```
main() {
  foo_factory_fp g = foo_factory_fp (*new foo_factory_impl() );

  try {
    naming<foo_factory>::rebind(domain_environment, \
            "village/services/the_factory", g);
  }
  except that {
    default {
      printf("The bind has failed.\n");
        }
  }
  cout << "The bind of object \"the_factory\" has been \
           successfully completed.\n";

  springlib::become_server();

}
```

The file contains the following *includes*:

- *foo_impl.h*, containing the class definitions and method prototypes.
- *naming/naming.h*, which provides an interface to the Spring naming system.
- *util/spring_lib.h*, which provides basic Spring definitions for all **.cc* files. This includes the method *springlib::become_server()*, which indeed causes the program to become a server, and wait to be contacted with method-invocations. While waiting, the server's address-space is not entirely dormant: it maintains an active thread for receiving invocations, and one or more others for initiating appropriate responses.

The *declare_naming* construct, shown above, is always used to declare the *type* of any object that is to be bound or resolved. In Spring, an interface determines the type of each object derived from it, and the interface-name—in this case, *foo_factory*—determines the type-name. (Further discussion of types and naming is provided in Chapter 4, *Interface Inheritance*.)

Spring provides two methods for publishing objects—*bind* and *rebind*. One possible advantage of *rebind* is that it does not generate an error if for some reason the name has already been bound; instead, it rebinds the name as specified in its own call. Additionally, as this example shows, it can be used when the specified name has not been bound.

The *naming* class, defined within *naming.h*, uses a C++ template to specify the name of the type to be bound, rebound, or resolved. The first argument to the *rebind* method (and indeed to the *bind* method) is the principal context that is given to any Spring domain when it is started up—this is always known as the *domain_environment*, and is roughly anaologous to the / directory in UNIX.

Various other contexts are automatically made accessible to the *domain_environment* (much as certain directories in UNIX might be mounted on /), and these include *machine* and *village*. When an object is bound within the *machine* context, only client domains running on the server's machine can access the published object. When an object is bound within *village*, any client domain within the currently established network of Spring machines can access the object. Both *machine* and *village* contexts contain a *services* child-context, within which object-names should reside. In this example, *"village/services/the_factory"* is specified as the second argument to *rebind*, indicating the compound name of the object.

The third argument to *rebind* is the variable containing the object reference that embodies the Spring object to be bound—the object reference is created in the normal way, via a **new* call on the constructor method of the implementation class, which is itself passed to the conversion operation for the *fp* class.

Once the specified object has been bound to a name and context, it remains there until the server process is terminated; the object-name binding is then itself removed.

Note that the simple name of the object (*the_factory*) is entirely arbitrary, and does not correspond to the type-name of the object.

The program contains an exception-handling routine, which generates a call to *printf* if the rebind fails.

3.1.5 Client Main Program Creation

The *client.cc* program wishes to use services provided by *server.cc*: thus, *client.cc* must access the object reference that *server.cc* has made available via the Spring name service. In this case, therefore, the client must know the name of the object it wishes to use—*the_factory*. It must also know the type of object to which it wishes to *resolve* the name—*foo_factory*. The object type must be declared at the head of the file by means of the *declare_naming* construct.

In fact, the code for *resolving* an object is similar to that for rebinding, which was shown previously. Once the object has been resolved, and is owned by the client as an object reference, the client can perform C++ method-invocations on the object reference, as previously demonstrated.

In the following program, the *foo_factory* is duly resolved, and its *create* method, which returns instantiated *foo* object references, is invoked by the client.

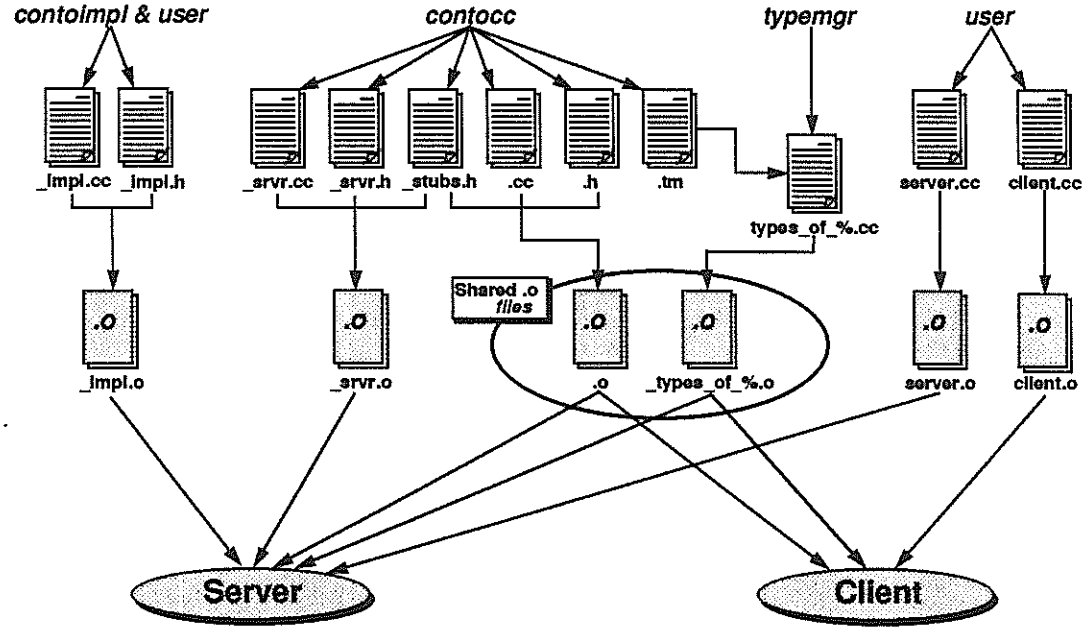
The file includes the client-side stubs interface provided by *foo.h*.

```
#include "foo.h"
#include <stream.h>
#include "naming/naming.h"
#include "util/basic.h"
#include "util/spring_lib.h"

declare_naming(foo_factory);

foo_factory_fp get_foo_factory();
void print (char *text, foo_fp ff);
foo_fp add(foo_fp f1, foo_fp f2, foo_factory_fp foofac);

main()
{
  foo_factory_fp g = get_foo_factory();
  foo_fp alpha = g->create(3);
  foo_fp beta = g->create(9);

  print("alpha: ", alpha);
  print("beta: ", beta);

  foo_fp gamma = add(alpha, beta, g);

  print("sum: ", gamma);

  return 0;
}

void print (char *text, foo_fp ff) {
  cout << text << ff->get_value() << "\n";
}

foo_fp add(foo_fp f1, foo_fp f2, foo_factory_fp foofac) {
  foo_fp f3 = foofac->create(f1->get_value() + f2->get_value() );
  return f3;
```

```
}

foo_factory_fp get_foo_factory() {
  foo_factory_fp g;
  for (int i = 0; i < 10; i++) {
    try {
      g = naming<foo_factory>::resolve(domain_environment, \
            "village/services/the_factory");
      break;
    } except that {
      naming_context_not_found {
    (void)printf("Waiting for \"the_factory\" to be published\n");
      }
    }
  }

    if (g == 0) {
      (void)printf("Could not look up \"the_factory\" \n");
      exit(1);
    }

  return g;
}
```

3.1.6 *Makefile Construction*

The following diagram summarizes the derivation of **.cc*, **.h*, and **.o* files from **.idl* interfaces, and indicates the roles played by these files in the creation of distributed client and server.



Note that both client and server require *foo.o* and *types_of_foo.o*—these are essentially client-side stubs files, which define both the object reference for foo and the foo type-definitions that the client needs. However, the files are also required by the server, which is itself expected to create and return *foo* object references.

Thus, the following Makefile might be used:

```
include VarRules
include $(OS)/Rules
include $(OS)/PT_Rules
include ContoccRules

ENV = foo.env
IDL_FILES = foo.idl
SERVER_CFILES = $(IDL_FILES:%.idl=%_srvr.cc) foo_impl.cc server.cc
SERVER_OFILES = $(SERVER_CFILES:%.cc=%.o)
```

```
SHARED_CFILES = $(IDL_FILES:%.idl=%.cc) \
        $(IDL_FILES:%.idl=types_of_%.cc)
SHARED_OFILES = $(SHARED_CFILES:%.cc=%.o)

PROGS = server client

all: $(PROGS) client.cc $(SERVER_CFILES) $(SHARED_CFILES)

server: $(SERVER_OFILES) $(SHARED_OFILES)
    $(SDLD) -o server $(SERVER_OFILES) $(SHARED_OFILES) \
        $(LIBRARIES)

client: client.o $(SHARED_OFILES)
    $(SDLD) -o client client.o $(SHARED_OFILES) $(LIBRARIES)

clean:
    $(RM) $(PROGS) *.o *~ core

superclean:
    $(RM) $(PROGS) *.o *~ core $(SHARED_CFILES) *_srvr.cc
```

The included file *ContoccRules* contains the rules for *contocc* and *typemgr*, which were shown in the previous example.

3.1.7 Compilation and Execution

Following successful compilation, presuming you are still running the *village* program, output is as follows:

```
sarum% server &
sarum% The bind of object "foo" has been successfully completed.
The bind of object "foo_factory" has been successfully completed.

sarum% client
alpha: 3
beta: 9
sum: 12
```

Naturally, following this call, the *server* process remains extant, and can be reused.

3.1.8 *Inter-Machine Process Communication*

In the program we have just analyzed, a server-implemented object is made available by means of the Spring name service. As we have seen, a specific context was cited, within which a name was bound to the object—this context was *village/services*.

The *village* represents a group of Spring machines that are associated with a given server. Any object that is bound within the *village* name space is accessible to processes on any machine within the village.

Thus, it is possible to run the server on one machine, and the client on another. This applies even in the case where one machine is running Spring, and the other SOUP: provided that client and server have been properly compiled on the respective operating systems on which they are to run, the interprocess communication will be successful.

Since a Spring machine currently requires a *tiptoe* connection to a UNIX host machine in order to be runnable, this intercommunication can be experimented with by any user who has a Spring machine. Simply run the server on the Spring machine, and the client on the UNIX hostmachine machine, or vice versa.

Note – You can substitute the context *machine* for *village*, and so make objects available for processes on your machine only. However, this only works for programs compiled and running under Spring—SOUP programs must continue to specify *village*. (This is due to an idiosyncracy of SOUP, whereby each SOUP process is in fact considered to be its own machine.)

Note that the Makefile requirements for SOUP are slightly different than those for Spring. Though this document intends only to consider source that is built and run on Spring, equivalent SOUP versions of practically all programs can be found in *~/tjh/programs*. In fact, the source in this hierarchy is divided between SOUP and Spring versions (though the Spring versions in *~/tjh/spring_progs* remain the most accurate and up-to-date). Thus, if you wish to run programs (or parts of programs) on SOUP, check this hierarchy for the appropriate Makefile elements.

3.1.9 Use of Client Implementation Libraries

In the previous example, the client was required to know the context and name of the server-provided object, and resolve the name in its own main program. While requiring the client to possess this knowledge may in some circumstances be entirely valid, it may also create risks. For example, if the server code should change, and the compound context under which the object is bound changes with it, the client itself would need to be rewritten and recompiled.

One solution to this problem is the creation of a client implementation library, using the *_clientimpl.cc* file provided by *contoimpl* (and possibly adding a *_clientimpl.h* file to accompany this). The server itself thus provides the code for resolving the object, and makes the **.o* file derived from *_clientimpl.cc* available to be linked against by all potential clients. Each client includes the *_clientimpl.h* file, and accesses the desired object by means of the function or method that the *_clientimpl.h* file declares.

An example of this is provided in the following locations:

```
~tjh/spring_progs/chap3/distrib_foo/lib_version
```

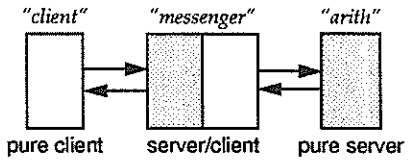
The program is identical to that examined above, except that object-access now occurs by means of a client implementation library. The *client* executable is now made as follows:

```
client: client.o foo_clientimpl.o $(SHARED_OFILES)
        $(DLD) -o client client.o foo_clientimpl.o $(SHARED_OFILES) \
            $(DYNAMIC_LIBS) -lg++
```

3.2 A Three-Domain Program

The following program extends the idea demonstrated in the previous section, and shows how server processes can function as interdependent clients. Differences between this program and the last are as follows:

- There are now three main programs—one for a pure client, one for a server/client, and one for a pure server. The client invokes methods on the server/client, which itself invokes methods on the pure server; thus, the programs form a chain of dependency.



- The pure server (*arith*) provides objects for addition, subtraction, and data passing.

  The server/client (*messenger*) accepts integers provided by the pure client, and uses these to initialize a data-carrying object, provided by the pure server.

  The server/client invokes an addition or subtraction method provided by the pure server, and in so doing, passes the data-carrying object to the pure server, where execution of the appropriate implementation method occurs.

  Within the data-carrying object, return values are then passed back from the pure server to the server/client; they are then passed back as integers to the pure client.

- Objects are bound and resolved in normal way. Note, however, that the server/client must resolve the pure server's objects within its *_impl.cc* file, as part of a method definition.

- Each server requires its own **.idl* and **.env* file. Thus, there are two of each.

The code for this example can be found in the following locations:

`~tjh/spring_progs/chap3/multi_distrib_foo`

In the following analysis, not all files are reproduced or described—only matters of particular note are afforded discussion.

3.2.1 *Interface Definition*

The Contract file for the pure server *arith* is as follows:

```
interface data {
    void set_integer (copy long a);
    long get_integer ();
```

```
};

interface data_factory {
    data create (copy long b);
};

interface addition {
    data add (consume data a, consume data b);
};

interface subtraction {
    data subtract (consume data a, consume data b);
};
```

A *factory* is defined to permit *data* object to be available to clients. The factory's *create* method returns a *data* object, as do the methods of *addition* and *subtraction*.

The Contract file for the client/server *messenger* is as follows:

```
interface messenger {
    long send_message (copy long a, copy long b, copy char ch);
};
```

The *send_message* method definition permits the passing of two integers and an operator sign, which is used by a client to determine whether *arith*'s *subtraction* or *addition* method is to be invoked by *messenger*.

3.2.2 Subcontract Specification

The **.env* file for *arith* is as follows:

```
$singleton: sc=singleton sc_file="subcontracts/singleton.h";
data: file = "arith.idl";
data_factory: file = "arith.idl";
addition: file = "arith.idl";
subtraction: file = "arith.idl";
data: $singleton;
data_factory: $singleton;
addition: $singleton;
subtraction: $singleton;
```

Each Spring object provided by *arith* uses the *singleton* subcontract. Thus, the first line of the **.env* file declares a macro that represents the necessary statement.

The following four lines state the Contract-file origin of the four objects. This statement will required by *messenger's* *.*env* file (see below).

The final four lines use the predefined macro to state that each object uses *singleton.*

The file for *messenger* is as follows;

```
import "arith.env";
messenger: sc=singleton sc_file="subcontracts/singleton.h";
```

Note that the file *arith.env* is imported into *messenger.env.* This will allow a single *.*env* file, containing all environment information pertinent to these domains, to be specified in the Makefile.

When multiple *.*env* files are merged by use of *import,* each subcontract specification must be made only in one of the files. Otherwise, *contocc* fails with an error message.

3.2.3 C++ Implementation of Defined Interfaces for the Pure Server

The file *arith_impl.cc* defines *add* and *subtract* methods that accept *data* objects as their argument, and return a *data* object as a result. The *add* method is as follows:

```
data_fp addition_impl::add(data_fp arg_a, data_fp arg_b)
    // interface ::addition
    // operation data add(data a data b )
{
  data_fp g;
  g = data_fp (*new data_impl() );
  g->set_integer(arg_a->get_integer() + arg_b->get_integer() );

  return g;
}
```

3.2.4 C++ Implementation of Defined Interfaces for Client/Server

Within the file *messenger _impl.cc,* the definition of the *send_message* method contains naming resolutions for the three objects that will be bound by the *arith* implementation—*data, addition,* and *subtraction.*

```
// messenger_impl.cc generated by contoimpl (3.7.0)
// from messenger.idl
// Tue Aug 17 15:01:17 1993
```

```
#include "messenger_impl.h"
#include "arith.h"
#include <stream.h>
#include "naming/naming.h"
#include "util/basic.h"
#include "util/spring_lib.h"

declare_naming(data_factory);
declare_naming(subtraction);
declare_naming(addition);

int messenger_impl::send_message(int arg_a, int arg_b, char ch)
    // interface ::messenger
    // operation long send_message(long a long b )
{
  subtraction_fp sbn;
  for (int i = 0; i < 10; i++) {
    try {
      sbn = naming<subtraction>::resolve(domain_environment, \
"village/services/subtraction");
      break;
    } except that {
      naming_context_not_found {
    cout << "Waiting for \"subtraction\" to be published.\n";
      }
    }
  }
  if (sbn == 0) {
    cout << "Could not look up \"subtraction\".\n";
    exit(1);
  }
```

*[similar naming resolutions for **addition** and **data_factory** appear here]*

```
  data_fp x = dfy->create(arg_a);
  data_fp y = dfy->create(arg_b);

  data_fp z;

  if (ch == '-') {
    z = (sbn->subtract(x, y));
```

```
  }
  else {
    z = (adn->add(x, y));
  }

  int answer = z->get_integer();

  return answer;
}
```

The *send_message* method uses the *data_factory* object in the expected way, producing *data* objects (*x* and *y*), which are then instantiated with the integer values provided by the client (*arg_a* and *arg_b*). In accordance with the operator-sign passed to it as an argument, the method passes the *data* object to either the *subtraction* or the *addition* object for computation.

3.2.5 *Pure Server and Client/Server Main Program Creation*

The file *arith_server.cc* binds its three objects in the expected way, then calls *become_server*. The file *messenger_server.cc* behaves likewise, binding the object *messenger*.

3.2.6 *Pure Client Main Program Creation*

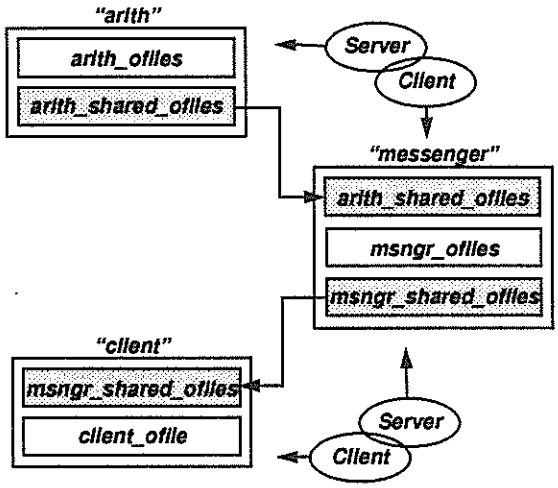
The file *client.cc* resolves the *messenger* object, and uses it to instantiate a messenger object reference *m*. It then accepts data from the command line, and uses this as arguments to messenger's *send_message* method:

```
int x, y, z;
char c;

cout << "Add or subtract? (Enter + or -): ";
cin >> c;

if (c != '+' && c != '-') {
  cout << "Invalid entry.\n";
    exit(1);
}

cout << "Enter first integer: ";
cin >> x;
cout << "Enter second: ";
cin >> y;
```

```
    z = m->send_message(x, y, c);

    cout << "The answer is " << z << "\n";

    return (0);
}
```

3.2.7 *Makefile Construction*

As in all previous examples, a client program must be linked against the **.o* and *types_of_*.o* files of its targeted server. The server itself also must be linked against these *shared* **.o* files, and additionally against its **_srvr.o*, **_impl.o*, and main **.o* files. Thus, for the three programs in this example, the dependencies can be represented as follows:



The corresponding Makefile is as follows:

```
include VarRules
include $(OS)/Rules
include $(OS)/PT_Rules
include ContoccRules

ENV = messenger.env

ARITH_IDL_FILES = arith.idl
ARITH_CFILES = $(ARITH_IDL_FILES:%.idl=%_srvr.cc)
ARITH_OFILES = $(ARITH_CFILES:%.cc=%.o) arith_server.o arith_impl.o
```

```
ARITH_SHARED_CFILES = $(ARITH_IDL_FILES:%.idl=%.cc) \
        $(ARITH_IDL_FILES:%.idl=types_of_%.cc)
ARITH_SHARED_OFILES = $(ARITH_SHARED_CFILES:%.cc=%.o)

MSGR_IDL_FILES = messenger.idl
MSGR_CFILES = $(MSGR_IDL_FILES:%.idl=%_srvr.cc)
MSGR_OFILES = $(MSGR_CFILES:%.cc=%.o) message_server.o \
        messenger_impl.o
MSGR_SHARED_CFILES = $(MSGR_IDL_FILES:%.idl=%.cc) \
        $(MSGR_IDL_FILES:%.idl=types_of_%.cc)
MSGR_SHARED_OFILES = $(MSGR_SHARED_CFILES:%.cc=%.o)

PROGS = arith_server message_server client

all: $(ARITH_CFILES) $(ARITH_SHARED_CFILES) $(MSGR_CFILES) \
        $(MSGR_SHARED_CFILES) $(PROGS)

arith_server: $(ARITH_OFILES) $(ARITH_SHARED_OFILES)
      $(SDLD) -o arith_server $(ARITH_OFILES) \
            $(ARITH_SHARED_OFILES) $(LIBRARIES)

message_server: $(MSGR_OFILES) $(MSGR_SHARED_OFILES) \
        $(ARITH_SHARED_OFILES)
     $(DLD) -o message_server $(MSGR_OFILES) $(MSGR_SHARED_OFILES) \
        $(ARITH_SHARED_OFILES) $(LIBRARIES)

client: client.o $(MSGR_SHARED_OFILES)
    $(DLD) -o client client.o $(MSGR_SHARED_OFILES) \
        $(LIBRARIES)

clean:
    $(RM) $(PROGS) *.o *~ core

superclean:
    $(RM) $(PROGS) *.o *~ core \
        types_of* *.tm *.stats \
        arith.cc arith.h arith_srvr.cc arith_srvr.h arith_stubs.h \
        messenger.cc messenger.h messenger_srvr.cc \
        messenger_srvr.h messenger_stubs.h
```

3.2.8 *Compilation and Execution*

The resulting session is as follows:

```
sarum% arith_server &
[3] 15325
```

```
sarum% The bind of "data_factory" has been successfully completed.
The bind of "subtraction" has been successfully completed.
The bind of "addition" has been successfully completed.

sarum% message_server &
[5] 15327
sarum% Bind of "messenger" successfully completed.

sarum% client
Add or subtract? (Enter + or -): +
Enter first integer: 5
Enter second: 3
The answer is 8
sarum% client
Add or subtract? (Enter + or -): -
Enter first integer: 3
Enter second: 9
The answer is -6
```

3.2.9 *Client Implementation Libraries*

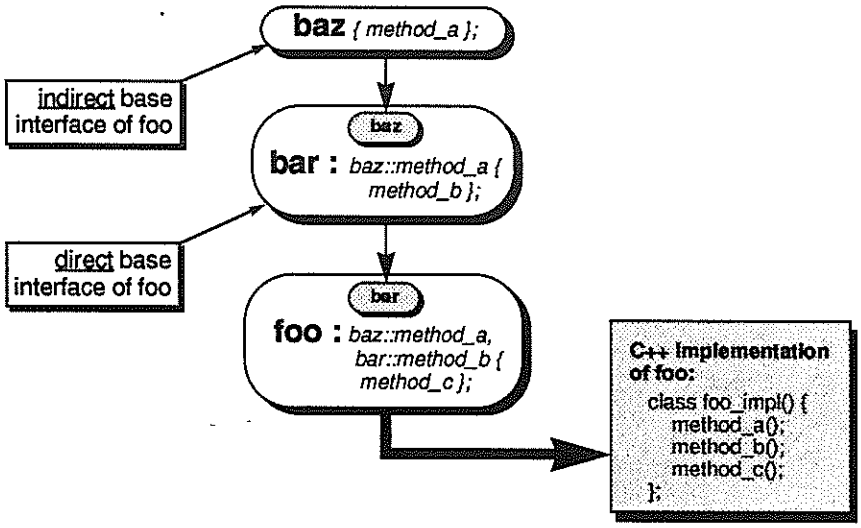
The following subdirectories provide an alternative version of this program, in which client implementation libraries are used for object-access:

```
~tjh/spring_progs/chap3/multi_distrib_foo/lib_version
```

Interface Inheritance 4

In the Contract language, each interface is considered to define a *type*, much as a class is considered to define a type in C++. Thus, interface *foo* defines type *foo*.

Contract supports multiple inheritance of interfaces, and thus of types. When interface *foo* inherits from interface *bar*, *foo* is termed the *derived* interface; *bar* is termed the *base* interface. Provided that the definition of *foo* has specifically referenced *bar* as an inherited interface, *bar* is termed a *direct* base interface of *foo*. Any interfaces that *bar* itself has inherited, and are not referenced in the definition of *foo*, are termed *indirect* base interfaces of *foo*:



An interface that is implemented as an object determines the *true type* of the object. As shown in the above illustration, the server that provides the implementation can implement all methods of whatever direct and indirect base interfaces are inherited by the *true type* interface.

Though Spring programming principally uses a system of *static typing*, and expects that all possible type checks will be performed at compile time, it nevertheless also supports a certain amount of *dyanmic typing*. This allows base objects to be treated as derived objects (so called *narrowing*), and derived objects as base (*widening*). Additionally, one base object can be treated as another (*traversing*), provided that both of these base objects are indeed inherited by a common more derived object—no direct or indirect inheritance relationship need exist between the base objects themselves. This *substitutability* of objects allows considerable freedom in the way objects are passed, bound, and resolved within Spring programs.

The examples in this chapter demonstrate how interface inheritance can be used in simple distributed programs. They additionally demonstrates use of *modules*, which are naming contexts used in Contract interfaces.

4.1 A Multiple-Inheritance Program

The sample code in this section extends the idea presented in the previous chapter, using a client and two servers to perform simple arithmetical operations by object passing.

The code for this example is in the following locations:

```
~tjh/spring_progs/chap4/inherit
```

4.1.1 Interface Definition

In this example, a single server implements interfaces contained by three separate Contract files—*data.idl*, *first_arith.idl*, and *second_arith.idl*—, and does so by programming against interfaces defined in still another file—*third_arith.idl*: this file contains interfaces that inherit all others.

The first of the files to be inherited, *data.idl*, appears as follows:

```
module data {

    interface data {
        void set_integer (copy long a);
```

```
        long get_integer ();
    };

    interface data_factory {
        data create (copy long b);
    };
```

The *module* construct is used to scope identifiers. Only one module can be used per file; the module can contain any number of interfaces. By convention, the file is named after the module, as is the case here. (However, elsewhere in this example, some files and modules are named differently, to help clarify some aspects of name-usage.)

The *data* and *data_factory* interfaces themselves are identical to those used in the three-domain examples in Chapter 3.

The other two Contract files to be inherited each provide methods for arithmetical operations: *first_arith.idl* provides methods for addition and subtraction, and is shown below:

```
import data;

module old_math {

    interface addition {
        data::data add (consume data::data a, consume data::data b);
    };

    interface subtraction {
        data::data subtract (consume data::data a, \
                                    consume data::data b);
    };

};
```

The module *data* is imported with the *import* construct—the exact location of the module *data* (the file *data.idl*) will thus need to be specified for the *contocc* compiler in an appropriate environment file. Within the module *old_math*, each reference to an interface from the module *data* must be correctly scoped, with the form *module-name::interface-name*. Thus, the *data* interface must appear as *data::data*.

The file *second_arith.idl* is similar in construction to *first_arith.idl*, and provides methods for multiplication and division, within a module named *new_math*.

The file *third_arith.idl* imports the contents of *data.idl, first_arith.idl,* and *second_arith.idl*:

```
import data;
import old_math;
import new_math;

module final_ops {

    interface final_math : old_math::addition, \
        old_math::subtraction, new_math::multiplication, \
            new_math::division {
    };

    interface final_data : data::data {
    };

    interface final_data_factory : data::data_factory {
    };

};
```

The modules *data, old_math,* and *new_math,* contained by the other files, are each imported into the current file. Within this file's own newly-defined module *final_ops,* new interfaces are defined (named *final_math, final_data,* and *final_data_factory),* which directly inherit the interfaces contained by the imported modules. The syntax for indicating inheritance is comparable to that used in C++ class inheritance. Here, the list of inherited interfaces follows the name of the derived interface, and is separated from it by a single colon. Each interface in the inheritance list is fully scoped, according to the module in which it resides. Interfaces in the list are separated from each other by a single comma.

The file *messenger.idl* defines an interface identical to that shown in the three-domain example in the previous chapter. The file imports the module *final_ops,* since the server associated with *messenger* will require access to the *arith*-defined objects within this module.

4.1.2 Subcontract Specification

For a server implementation to be based on *third_arith,* a single **.env* file must be created, which associates an appropriate subcontract with each interface either defined or inherited by *third_arith.idl*:

```
$singleton: sc=singleton sc_file="subcontracts/singleton.h";
data: file = "data.idl";
old_math: file = "first_arith.idl";
new_math: file = "second_arith.idl";
final_ops: file = "final_arith.idl";
data::data: $singleton;
data::data_factory: $singleton;
old_math::addition: $singleton;
old_math::subtraction: $singleton;
new_math::multiplication: $singleton;
new_math::division: $singleton;
final_ops::final_math: $singleton;
final_ops::final_data: $singleton;
final_ops::final_data_factory: $singleton;
```

Line one creates a macro for specifying the *singleton* subcontract. Lines two to five specify the Contract file locations of the *data, old_math, new_math,* and *final_ops* modules. The remaining lines specify the subcontracts for each interface in *final_ops,* including those inherited from the other modules.

4.1.3 C++ Implementation of Defined Interfaces

The C++ implementation class- and method-definitions for *third_arith* are contained in the files *third_arith_impl.h* and *third_arith_impl.cc.* The file *third_arith_impl.cc* contains the following method definition:

```
data_fp final_ops_final_math_impl::add(data_fp arg_a, data_fp \
                                       arg_b)
    // interface ::old_math::addition
    // operation data::data add(data::data a data::data b )
{
  data_fp g;
  g = data_fp (*new final_ops_final_data_impl() );
  g->set_integer(arg_a->get_integer() + arg_b->get_integer() );

  return g;
}
```

This fragment indicates the effect of the use of *modules* on C++ *contoimpl*-generated implementation names. Here, the implementation class takes its name from the combination of the module and interface names—thus, the Contract-specified operation *final_ops::final_math* has become *final_ops_final_math_impl.*

The fragment also shows how Contract inheritance (and implementation of a derived interface) does not change the name or identity of fat pointers produced by base interfaces: the *add* method takes *data* fat pointers as arguments, and also returns a *data* fat pointer; additionally, within the body of this method, a *data* fat pointer is created (by a call to the constructor for the *final_ops_final_data_impl* implementation class). Thus, the name and identity of the fat pointer defined from the *base* interface *data* remain unchanged, even though the current implementation class is based on the *derived* Contract interface, *final_data*.

The file *third_arith_impl.cc* also contains a factory *create* method for client use, which is as follows:

```
data_fp final_ops_final_data_factory_impl::create(int arg_b)
    // interface ::data::data_factory
    // operation data create(long b )
{
  data_fp k;
  k = data_fp (*new final_ops_final_data_impl() );
  k->set_integer(arg_b);

  return k;
}
```

Again, the method manipulates and produces *data* fat pointers.

Note also the implementation class *final_ops_final_math_impl*, which is defined within *third_arith_impl.h*:

```
class final_ops_final_math_impl:
    public final_ops_final_math_srvr
{
public:
    virtual data_fp add(data_fp arg_a, data_fp arg_b);
    virtual data_fp subtract(data_fp arg_a, data_fp arg_b);
    virtual data_fp multiply(data_fp arg_a, data_fp arg_b);
    virtual data_fp divide(data_fp arg_a, data_fp arg_b);
};
```

The methods derived from *old_math* and *new_math* are all afforded C++ implementation within the same class: thus, their Contract inheritance path is invisible within the server implementation.

Expectedly, the *messenger_impl.h* file contains the C++ class definition for *messenger_impl*. The file *messenger_impl.cc* contains the following *includes* and naming declarations:

```
#include "first_arith.h"
#include "second_arith.h"
#include "data.h"
#include <stream.h>
#include "naming/naming.h"
#include "util/basic.h"
#include "util/spring_lib.h"
#include "messenger_impl.h"

declare_naming(data_data_factory);
declare_naming(old_math_addition);
declare_naming(old_math_subtraction);
declare_naming(new_math_multiplication);
declare_naming(new_math_division);
```

Though the objects it needs will be published by a server-implementation of *third_arith.idl*, the client program will remain a client of the *first_arith, second_arith,* and *data* interfaces, which are providing the specification of the basic functionality. Thus, the **.h* files created by running *contocc* on each of these base interfaces are necessarily included in this client implementation file. However, the **.h* file for the implementation of *third_arith.idl* is not required, since no objects based exclusively on *third_arith.idl* are being implemented.

A naming declaration is provided for each object-type that will be bound by the server and resolved by the client. As previously demonstrated, the type-name of each object is derived from the conflation of its module-name and interface-name. Thus, the *data_factory* object becomes *data_data_factory.*

Elsewhere in *messenger_impl.cc,* the declared objects are resolved in the usual way. Objects' methods are then invoked, depending on information supplied from the command line, via *client.cc.*

4.1.4 Main Program Creation

The object-types resolved above in *messenger_impl.cc* are bound in *third_arith_server.cc,* as indicated by the following fragment. Note in particular the way in which fat pointers are created:

```
                        .
                        .
declare_naming(old_math_subtraction);
declare_naming(new_math_multiplication);
                        .
                        .
```

```
main() {
                        .
                        .
    old_math_addition_fp m = old_math_addition_fp \
        (*new final_ops_final_math_impl() );

                        .
                        .
    naming<old_math_addition>::rebind(domain_environment, \
        "village/services/old_math_addition", m);
                        .
                        .
    new_math_multiplication_fp o = new_math_multiplication_fp \
        (*new final_ops_final_math_impl() );
                        .
                        .
    naming<new_math_multiplication>::rebind(domain_environment,\
                    "village/services/new_math_multiplication", o);
                        .
                        .
  springlib::become_server();
}
```

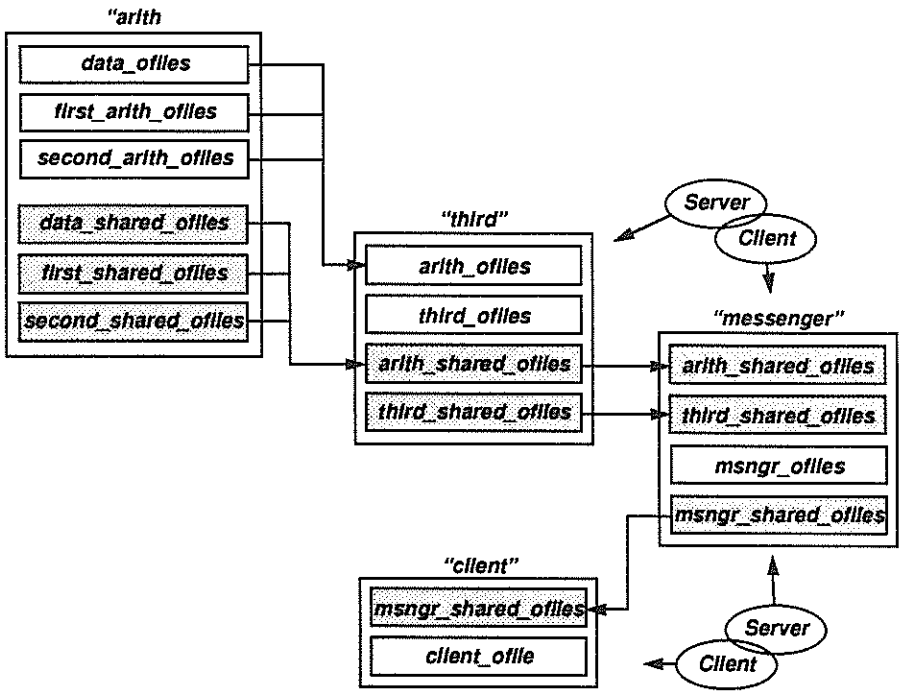
In this fragment, the constructor for *final_ops_final_math_impl*, which is the *third_arith*-based implementation class that inherited all methods from the other **.idl* files, is called twice. On the first occasion, this ultimately results in the creation of an *old_math_addition* fat pointer, and on the second, in the creation of a *new_math_multiplication* fat pointer. Other resolutions are used elsewhere in this file to create fat pointers for the other objects specified by the base interfaces. Thus, a single C++ implementation class is shown to use its constructor to support fat pointers for all the inherited objects that it implements.

The file *messenger_server.cc* resolves each of the object-names that have been bound by *third_arith_server.cc*. The code in *client.cc* behaves almost identically to the version in the three-domain example shown in the previous chapter.

4.1.5 Makefile Construction

Once again, each client program must be linked against the **.o* and *types_of_*.o* files of its targeted server. Each server itself also must be linked against these *shared* **.o* files, and also against its **_srvr.o*, **_impl.o*, and main **.o* files.

In the current example, this includes the **.o* files for the inherited interfaces, *data.idl, first_arith.idl,* and *second_arith.idl.* The dependencies can be illustrated as follows:



This is further demonstrated by the Makefile, reproduced below:

```
include VarRules
include $(OS)/Rules
include $(OS)/PT_Rules
include ContoccRules

ENV = messenger.env

ARITH_IDL_FILES = data.idl first_arith.idl second_arith.idl
ARITH_CFILES = $(ARITH_IDL_FILES:%.idl=%_srvr.cc)
ARITH_OFILES = $(ARITH_CFILES:%_srvr.cc=%_srvr.o)
ARITH_SHARED_CFILES = $(ARITH_IDL_FILES:%.idl=%.cc) \
        $(ARITH_IDL_FILES:%.idl=types_of_%.cc)
ARITH_SHARED_OFILES = $(ARITH_SHARED_CFILES:%.cc=%.o)

THIRD_ARITH_IDL = third_arith.idl
```

```
THIRD_CFILES = $(THIRD_ARITH_IDL:%.idl=%_srvr.cc) \
        third_arith_impl.cc third_arith_server.cc
THIRD_OFILES = $(THIRD_CFILES:%.cc=%.o)
THIRD_SHARED_CFILES = $(THIRD_ARITH_IDL:%.idl=%.cc) \
        $(THIRD_ARITH_IDL:%.idl=types_of_%.cc)
THIRD_SHARED_OFILES = $(THIRD_SHARED_CFILES:%.cc=%.o)

ALL_ARITH_SHARED_CFILES = types_of_third_arith.cc third_arith.cc \
        $(ARITH_SHARED_CFILES)
ALL_ARITH_SHARED_OFILES = $(ARITH_SHARED_OFILES) \
        $(THIRD_SHARED_OFILES)
ALL_ARITH_OFILES = $(ARITH_OFILES) $(THIRD_OFILES)

MSGR_IDL = messenger.idl
MSGR_CFILES = $(MSGR_IDL:%.idl=%_srvr.cc) messenger_impl.cc \
        message_server.cc
MSGR_OFILES = $(MSGR_CFILES:%.cc=%.o)
MSGR_SHARED_CFILES = $(MSGR_IDL:%.idl=%.cc) \
        $(MSGR_IDL:%.idl=types_of_%.cc)
MSGR_SHARED_OFILES = $(MSGR_SHARED_CFILES:%.cc=%.o)

PROGS = third_arith_server message_server client

all: $(ALL_ARITH_SHARED_CFILES) $(MSGR_SHARED_CFILES) $(PROGS)

third_arith_server: $(ALL_ARITH_SHARED_OFILES) $(ALL_ARITH_OFILES)
    $(SDLD) -o third_arith_server $(ALL_ARITH_SHARED_OFILES) \
        $(ALL_ARITH_OFILES) $(LIBRARIES)

message_server: $(MSGR_SHARED_OFILES) $(MSGR_OFILES) \
        $(ALL_ARITH_SHARED_OFILES)
    $(SDLD) -o message_server $(MSGR_SHARED_OFILES) $(MSGR_OFILES) \
        $(ALL_ARITH_SHARED_OFILES) $(LIBRARIES)

client: client.o
    $(SDLD) -o client client.o $(MSGR_SHARED_OFILES) $(LIBRARIES)

clean:
    $(RM) $(PROGS) *.o core

superclean:
    $(RM) $(PROGS) *.o core *.stats *.tm types_of_* \
        data.h first_arith.h second_arith.h messenger.h \
        third_arith.h *_stubs.h *srvr.* \
        data.cc first_arith.cc second_arith.cc third_arith.cc \
        *~
```

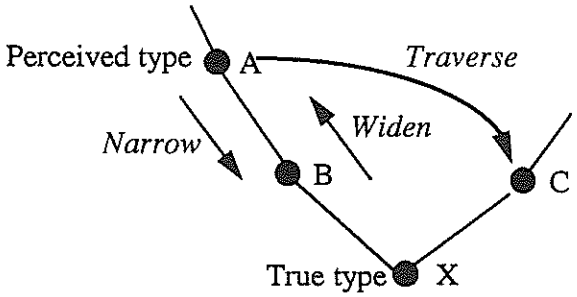
4.1.6 Execution

```
sarum% my_arith_server &
[3] 6613
sarum% The bind of "data_data_factory" has been successfully
completed.
The bind of "old_math_addition" has been successfully completed.
The bind of "old_math_subtraction" has been successfully completed.
The bind of "new_math_multiplication" has been successfully
completed.
The bind of "new_math_division" has been successfully completed.

sarum% message_server &
[5] 6616
sarum% Bind of "messenger" successfully completed.

sarum% client
Add, subtract, multiply, or divide? (Enter +, -, *, or /): *
Enter first integer: 3
Enter second: 6
The answer is 18
```

4.2 Substitutability

Though Spring programming principally uses a system of *static typing*, and expects that all possible type checks will be performed at compile time, it nevertheless also supports a certain amount of *dynamic typing*. This allows base objects to be treated as derived objects (so called *narrowing*), and derived objects as base (*widening*). Additionally, one base object can be treated as another (*traversing*), provided that both of these base objects are indeed inherited by a common more derived object—no direct or indirect inheritance relationship need exist between the base objects themselves. This *substitutability* of objects allows considerable freedom in the way objects are passed, bound, and resolved within Spring programs.

The nature of object substitutability is shown in the following illustration:



This illustration assumes the existence of a derived interface, *X*, which inherited interface *B*, which itself inherited interface *A*. The derived interface *X* also inherited interface *C*, which is otherwise not connected with *A* or *B*.

X, then, is assumed to have been implemented as an object. Some client code requires an object of type *A*, but is passed the object of type *X* instead. This does not cause a problem for the client code, since *widening* takes place dynamically, and the client is thus permitted to perceive the passed object as one of type *A*. (Note that *widening* is implicitly specified whenever a derived object is substituted for a base—no explicit call or invocation is required from the holder of the object.)

At some other point during the program, the client may wish to pass the object on to another client or server, which requires an object of type *B*. Again, this can legitimately, provided that *narrowing* is explicitly performed by the client that currently holds the object: this will indeed allow the object to be perceived as one of type B. (Note, then, that *narrowing*, unlike *widening*, must be explicitly performed by the holder of the object—all fat pointers have a defined static method that can be called to effect this.)

Finally, the object may need to be passed again, this time to a process that requires an object of type C. Even though there is no direct relationship between *A* and *B* on the one hand, and C on the other, the object can legitimately be passed as one of type C, provided that *traversing* is explicitly performed by the holder of the object. Again, all fat pointers have a static method that effects this.

Essentially, therefore, known existence of a *true type* object permits substitutability of any *base type* object.

4.2.1 Some Examples of Substitutability in Object-Name Resolution

In the preceding example (in Section 4.1), to create various fat pointers corresponding to base interfaces, a single C++ constructor method was called: this was the constructor for the implementation class of the most derived interface. The following code fragments from *third_arith_server.cc* can be used to demonstrate this issue again:

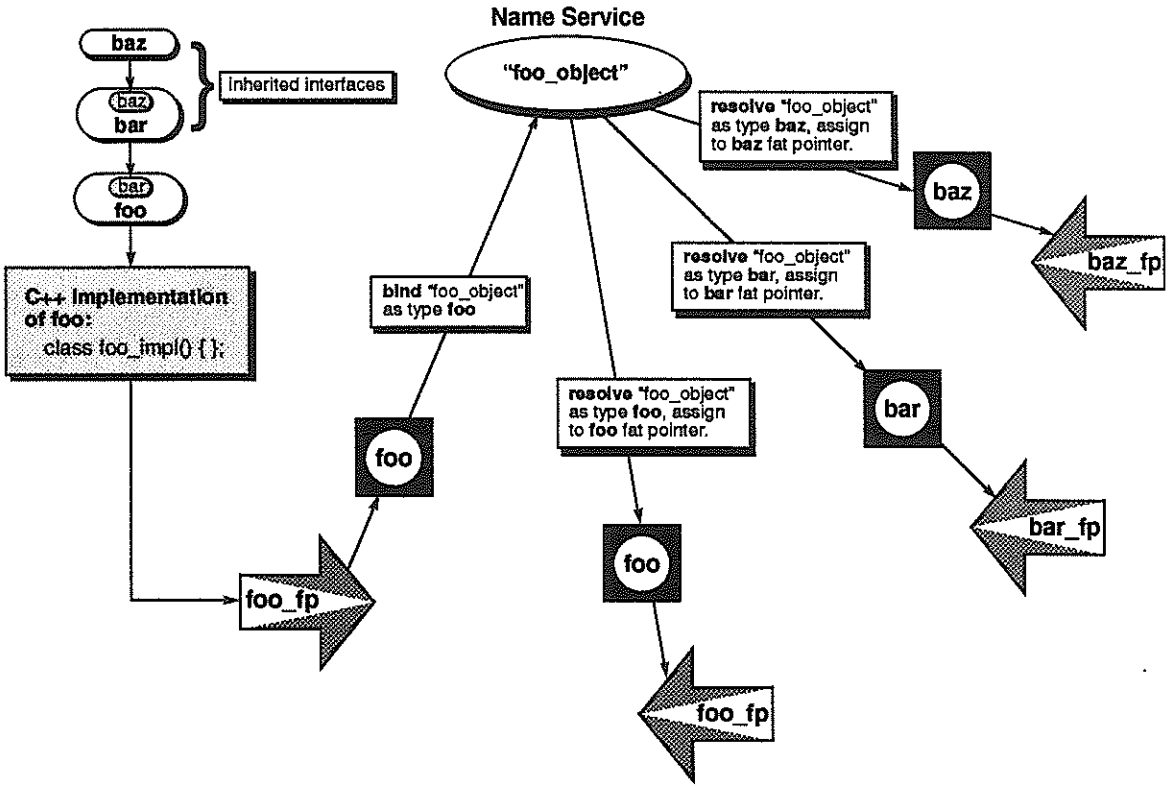
```
                    .
                    .
old_math_addition_fp m = old_math_addition_fp \
      (*new final_ops_final_math_impl() );
                    .
                    .
  naming<old_math_addition>::rebind(domain_environment, \
      "village/services/old_math_addition", m);
                    .
                    .
new_math_multiplication_fp o = new_math_multiplication_fp \
      (*new final_ops_final_math_impl() );
                    .
                    .
  naming<new_math_multiplication>::rebind(domain_environment, \
      "village/services/new_math_multiplicaton", o);
                    .
                    .
```

As this shows, the constructor *final_ops_final_math_impl()* was called repeatedly to produce different fat pointers, each corresonding to a base interface. The type of each base interface was specifically used in binding the object to a name.

Each published base object was then resolved by the client according to its published type. Thus, if an object was of type *new_math_multiplication*, it was specifically resolved as such. This is shown by the next fragment, taken from *messenger_impl.cc:*

```
                    .
                    .
declare_naming(new_math_multiplication);
                    .
                    .
  new_math_multiplication_fp q;

  q = naming<new_math_multiplication>::resolve(domain_environment,\
                "village/services/new_math_multiplication");
```

However, Spring's dynamic typing system permits more flexible means of resolving objects. For example, when a server binds the name of a derived object, the client can resolve this name either into the derived object, or into any base object:



This form of resolution is exemplified by the code example contained in the following locations:

```
~tjh/spring_progs/chap4/inherit/subst/naming/example1
```

This is a variant of the interface inheritance program used in the previous section. The server-implementation file *third_arith_server.cc* contains the following code, which binds a name for the derived object implemented from *third_arith.idl*:

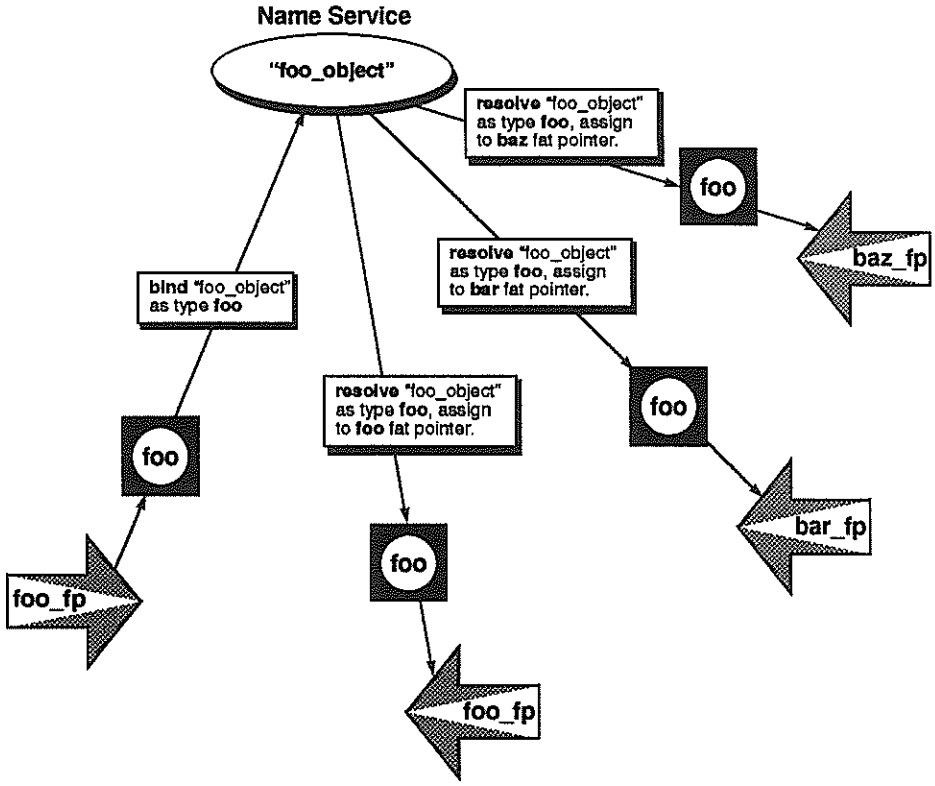
```
                    .
declare_naming(final_ops_final_math);
                    .
                    .
final_ops_final_math_fp m = final_ops_final_math_fp \
    (*new final_ops_final_math_impl() );
                    .
                    .
naming<final_ops_final_math>::rebind(domain_environment, \
            "village/services/math:, m);
                    .
                    .
```

Once the derived object's name *math* has been bound, the client of the object, *messenger_impl.cc,* is free to resolve the name to any one of its required base object types:

```
                    .
                    .
declare_naming(new_math_multiplication);
                    .
                    .
new_math_multiplication_fp q;
                    .
                    .
q = naming<new_math_multiplication>::resolve(domain_environment, \
            "village/services/math");
                    .
                    .
```

Here, the client declares that a *new_math_multiplication* object will be used; it then creates an appropriate fat pointer and, by means of a *naming* call, resolves the name *math* to a *new_math_multiplication* object. (Elsewhere in the file, the client's other required base objects are resolved in the same way.)

Still another way of resolving the name of a derived object involves the client's obtaining a copy of the derived object itself, then using the object within the implementation environment to manufacture fat pointers of the required base object types. This is indicated by the following diagram, which again assumes that interface *foo* inherits *bar* and *baz*:



This form of object-name resolution is exemplified by the code in the following locations:

`~tjh/spring_progs/chap4/inherit/subst/naming/example2`

This code is a further variant of the three-domain inheritance program. Here, *third_arith_server.cc* binds a single object-name, *final_ops_final_math*, to represent the Spring object created from the C++ class *final_ops_final_math_impl*: this class is the C++ implementation of the interface *final_math*, which is contained in *third_arith.idl*, and inherits all arithmetical operations of the base interfaces *old_math* and *new_math*:

```
declare_naming(final_ops_final_math);
                        .
                        .
  final_ops_final_math_fp m = final_ops_final_math_fp \
    (*new final_ops_final_math_impl() );
                        .
                        .
    naming<final_ops_final_math>::rebind(domain_environment, \
                  "village/services/final_ops_final_math", m);
                        .
                        .
  springlib::become_server();
                        .
                        .
```

When the client-implementation file *messenger_impl.cc* resolves the name *final_ops_final_math*, it specifies that the object should indeed be of type *final_ops_final_math*. However, it then assigns the object to fat pointer variables that represent the base types from which *final_ops_final_math* was derived:

```
                        .
                        .

#include "data.h"
#include "first_arith.h"
#include "second_arith.h"
#include "third_arith.h"          // Note that this is now needed.
                        .
                        .
declare_naming(final_ops_final_math);
                        .
                        .
  old_math_addition_fp o;

    o = naming<final_ops_final_math>::resolve(domain_environment,
"village/services/final_ops_final_math");
                        .
                        .
   new_math_multiplication_fp q;
    q = naming<final_ops_final_math>::resolve(domain_environment,
"village/services/final_ops_final_math");
                        .
                        .
```

Note that, unlike earlier versions of *messenger_impl.cc*, the present version must include *third_arith.h*, since it is using the *final_ops_final_math* object, created from an interface within *third_arith.idl*. (Previous versions only used base objects, which came from *data.idl*, *first_arith.idl*, and *second_arith.idl*.)

Ultimately, then, provided that the server has bound a derived object to a name, any base or derived object can be resolved from it. If there is no particular need specifically to provide a base object, the simplest course is to resolve the derived object as such, and invoke all base methods upon it. This is demonstrated in the following locations:

`~tjh/spring_progs/chap4/inherit/subst/naming/example3`

Here, the file *third_arith_server.cc* provides a name-binding for the *final_ops_final_math* object—this is the derived object that inherits all four interfaces contained by *first_arith.idl* and *second_arith.idl*:

```
                        .
                        .
final_ops_final_math_fp m = final_ops_final_math_fp \
    (*new final_ops_final_math_impl() );
                        .
                        .
    naming<final_ops_final_math>::rebind(domain_environment, \
        "village/servies/final_ops_final_math", m);
                        .
                        .
```

In the client-file *messenger_impl.cc*, the published name is duly resolved to a fat pointer for the derived object itself:

```
                        .
                        .
final_ops_final_math_fp o;
                        .
                        .
o = naming<final_ops_final_math>::resolve(domain_environment, \
    "village/services/final_ops_final_math");
                        .
                        .
```

Following name-resolution, all required operations—as provided in the base interfaces in *first_arith.idl* and *second_arith.idl*—are invoked directly on the derived object:

```
                        .
                        .
switch (arg_ch) {
```

```
case '+':
    z = o->add(x, y);
    break;
case '-':
    z = o->subtract(x y);
    break;
case '*';
    z = o->multiply(x, y);
    break;
case '/':
    z = o->divide(x, y);
default:
    cout << "unknown operator.\n";
    break;
}
```

.
.

4.2.2 *Some Examples of Widening, Narrowing, and Traversing*

As previously described, *widening, narrowing,* and *traversing* allow substitutability of objects to occur in argument passing and method invocation. *Widening* is specified implicitly whenever a derived object is substituted for a base. *Narrowing* and *traversing* must be specified explicitly by the holder of the object, using Spring-defined static methods that are generic to all fat pointers.

An example of *widening* can be found in the following locations:

`~tjh/spring_progs/chap4/inherit/subst/passing/widen`

In further extending the multiple inheritance example that has been previously examined, this new version adds an additional process, named *calc_server.* This process represents a gratuitous level of indirection, whereby the program *message_server* no longer actually invokes the object-methods made available by *arith_server*: instead, it passes all objects required for calculation to the new *calc_server,* which accordingly handles the invocations.

Widening is thus demonstrated in the file *messenger_impl.cc,* which obtains the object provided by *calc_server.cc,* and invokes the methods defined in *calc_impl.cc.* These method-invocations require that the base objects implemented by *third_arith_impl.cc* be passed as arguments. The following fragment indicates the method defined by *calc_impl.cc* for performing division:

```
data_fp calc_impl::do_divide(new_math_division_fp arg_h, data_fp \
                                        arg_i, data_fp arg_j)
```

```
{
    data_fp dfp = arg_h->divide(arg_i, arg_j);
    return(dfp)
}
```

However, in the client-file *messenger_impl.cc*, it is the most derived object—of type *final_ops_final_math*—that is passed to *do_divide* and to the other defined methods:

```
final_ops_final_math_fp fofm;
                        .
                        .
fofm = naming<final_ops_final_math>::resolve(domain_environment, \
           "village/services/final_ops_final_math");
                        .
                        .
switch (arch_ch) {
case '+':
    z = clc->do_add(fofm, x, y);
    break;
case '-':
    z = clc->do_subtract(fofm, x, y);
    break;
case '*':
    z = clc->do_multiply(fofm, x, y);
    break;
case '/':
    z = clc->do_divide(fofm, x, y);
    break;
default:
    cout << "Unknown operator.\n";
    break;
}
                        .
                        .
```

Widening is performed dynamically, and the derived object is perceived by *calc* in each case to be the base object appropriate for method-invocation.

By contrast, whereas widening needs no explicit call, *narrowing* and *traversing* must indeed by explicitly specified by the process that holds the object. An example of both narrowing and traversing can be found in the following locations:

~tjh/spring_progs/chap4/inherit/subst/passing/nrw_and_trvrse

This program again uses the indirection supplied by *calc_server*. Additionally, the program makes a modification to the interface *final_ops::final_math*, which itself inherits the methods provided by the *addition, subtraction, multiplication,* and *division* interfaces—it adds a new method to the *final_math* interface, which returns the square of an integer. The interface (located in *third_arith.idl*) thus appears as follows:

```
interface final_math : old_math::addition, old_math::subtraction,
                       new_math::multiplication, new_math::division {
    data::data square(consume data::data a);
    };
```

Along with the inherited methods, the new method is implemented in the server implementation file *third_arith_impl.cc*. As in the last example, *messenger_impl.cc* does not itself invoke these methods—instead, it passes the appropriate objects to *calc_server*, which duly performs the invocations:

```
old_math_addition_fp o;
                          .
                          .
o = naming<final_ops_final_math>::resolve(domain_environment, \
            "village/services/final_ops_final_math");
                          .
                          .
switch (arch_ch) {
case '+':
    z = clc->do_add(o, x, y);
    break;
case '-':
    old_math_subtraction_fp p;
    p = old_math_subtraction::_traverse(o);
    z = clc->do_subtract(p, x, y);
    break;
case '*':
    new_math_multiplication_fp q;
    q = new_math_multiplication::_traverse(o);
    z = clc->do_multiply(q, x, y);
    break;
case '/':
    new_math_division_fp r
    r = new_math_division::_traverse(o);
    z = clc->do_divide(r, x, y);
    break;
case '^':
```

```
    final_ops_final_math_fp s;
    s = final_ops_final_math::_narrow(o);
    z = clc->do_square(s, x);
default:
    cout << "Unknown operator.\n";
    break;
}
```

Though the object provided by name resolution is of the most derived type (*final_ops_final_math*), it is initially assigned to a base type fat pointer (*old_math_addition_fp*). This fat pointer can be passed unchanged to the *do_add* method of the *calc* object. However, if any other base type object is required for the *do_add* method, the perceived *old_math_addition* object must be changed to a different base type through use of the static method *_traverse*, which is defined for all fat pointers. The *_traverse* method called is that of the class to which the existing object is to be traversed. Its sole argument is the fat pointer for the existing object. Its return value is the fat pointer for the object to which traversal has occurred.

In the above fragment, in the penultimate *case* of the *switch* statement, the static method *_narrow* is invoked. The fat pointer of the perceived base object is thus transformed to a fat pointer for the derived object, and this is used to invoke the *do_square* method of the *calc* object.

Note that, once again, these substitutions are only possible due to the initial presence of the most derived object (*final_ops_final_math*), which was used in name binding by the server. The client should only perform these substitutions if it knows that the object it truly possesses is of the appropriate derived type.

4.3 *Pure and Concrete Type Inheritance*

In Spring, an interface determines the *type* of each object derived from it, and the interface-name determines the type-name. As indicated in the previous section, types can inherit one another, and the type of an object is defined by its *most-derived interface*—that is, the interface at the bottom of the inheritance chain, which thus contains all the functionality described by the base interfaces.

In terms of inheritance, two kinds of type are considered to exist: *pure* and *concrete*. A *pure* type is one that is intended as the basis for inheritance. A pure type may itself inherit from other pure types. Thus, interface *book* should probably inherit the pure types *cover, pages,* and *text,* and should itself be a pure type capable of being inherited by types *dictionary, diary,* and *novel.*

A *concrete* type is one that gathers together the properties of a number of pure types, but is not itself intended as a basis for inheritance. Thus, the inherited combination of pure types is appropriate for a single anticipated implementation, but is not itself offered as a pure type to be inherited by new, additional subtypes.

This distinction helps to prevent the creation of an inheritance system that contains an excessive number of mandatorily inherited types—many types may be required for only occasional or optional implementation uses, and may thus conflict with other, similarly optional types.

For example, a given implementation of *book* might require a color to be specified for its cover—say, *blue* or *red*. But while it is quite possible for *book* to inherit both interfaces, a given implementation of *book* will require the functionality of one of them at most. Moreover, if an implementor of *book* should ideally have a much broader choice of color, many hundreds of color interfaces would need to be inherited; the *book* interface would thus become bloated with functionality unneeded by any given implementor. And, if *book* itself were a pure type, and were duly inherited by, say, *dictionary,* the problem would be passed even further down the line of inheritance.

Thus, types such as *red* and *blue* can more usefully be considered as types that are *pure,* but are intended to be inherited only for specific implementations. Thus, while *book* might be inherited by *dictionary,* neither *book* nor *dictionary* would itself inherit any color interface as a pure type. Instead, a *concrete* type named *red_dictionary* could be created: this type would inherit the *dictionary* pure type, and would be identical to it, except that it would choose to inherit the *red* type. Thus, no other implementor of *dictionary* would be compelled to inherit the *red* interface.

Spring provides functionality that allows such *concrete inheritance* to occur. This is demonstrated by the code in the following locations:

```
~tjh/spring_progs/chap4/my_inherit
```

The program again modifies the three-domain inheritance example, which was first shown in Section 4.1.1. This time, the inherited interfaces in *first_arith.idl* and *second_arith.idl* are considered to be pure types requiring implementation through a concrete type. Thus, instead of subclassing the interfaces by creating pure types (such as those formerly found in *third_arith.idl*), the program does so by creating concrete types, contained in the file *my_arith.idl*.

The use of *my_* is an accepted idiom for indicating that the interfaces within a Contract file represent concrete types, created for a unique implementation circumstance, and not to be used as the basis for further inheritance. Additionally, interfaces created within a *my_* Contract file may not be directly programmed against by clients. Instead, clients must program against the base interfaces. Within the server, Spring objects may be created from the derived interfaces, and these may be published via the name service. However, the client may resolve the object-names only to their base objects—the derived objects cannot thus be accessed, and any attempts to access them produce only an error at link-time.

The contents of *my_arith.idl* are as follows:

```
import data;
import old_math;
import new_math;

module my_ops {

    interface my_math : old_math::addition, \
        old_math::subtraction,  new_math::multiplication, \
            new_math::division {
    };

    interface my_data : data::data {
    };

    interface my_data_factory : data::data_factory {
    };

};
```

There is, in fact, no substantial difference between the contents of this file and those of the originally demonstrated *third_arith.idl*. Modules and interfaces now have *my_* prepended to their names, but this is a superficial change, and has no automatic effect on compilation.

Within the server implementation file *my_arith_impl.h*, the appropriate C++ implementation classes are defined: these include *my_ops_my_math_impl*, which contains methods corresponding to the operations inherited by the *my_math* interface, shown above. In the file *my_arith_server.cc*, the name *math* is associated with the type *my_ops_my_math*, and is bound under the name *math*:

```
                    .
                    .
my_ops_my_math_fp m = my_ops_my_math_fp \
        (*new my_ops_my_math_impl() );
                    .
                    .
    naming<my_ops_my_math>::rebind(domain_environment, \
        "village/services/math", m);
```

Note that as an alternative, the constructor for the derived object could have been used to return the fat pointers of each base object, which could then themselves have been individually name-bound.

In the client file *messenger_impl.cc*, the name of the derived *my_ops_my_math* object (*math*) is resolved to the corresponding base objects. For example:

```
                    .
                    .
#include "first_arith.h"
                    .
                    .
declare_naming(old_math_addition);
                    .
                    .
old_math_addition_fp o;
                    .
                    .
    o = naming<old_math_addition>::resolve(domain_environment, /
            "village/services/math")
                    .
                    .
```

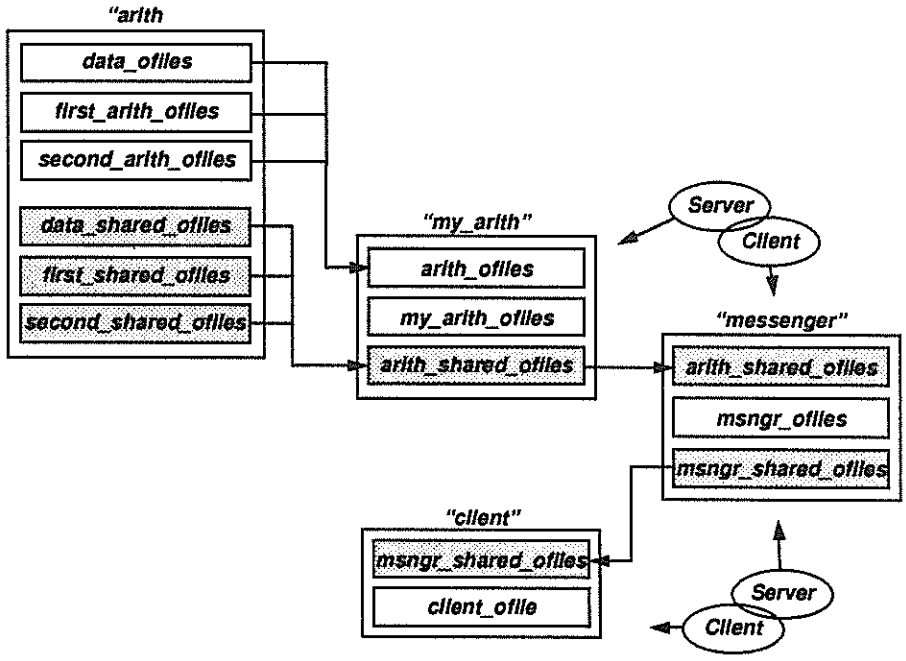
Again, this is the only legitimate way of resolving the name of a concrete type. The derived object itself—*my_ops_my_math*—cannot be resolved.

In construction, then, the program is almost identical to the first version we considered, which used the pure types created by *third_arith.idl*. The principal difference comes in compilation of the *my_arith.idl* interfaces, in which *contocc* is relinquished in favor of *concrete_stubs*.

The *concrete_stubs* program generates only the stubs for the server side. Thus, the stubs files that usually support the client side—and are indeed usually shared between client and server—, **.h*, **.cc*, *types_of_*.h*, and *types_of_*.cc*, are not generated. Instead, the information that continues to be required by the server (so that, for example, the derived object can be created as a fat pointer and then published) is compiled into the **_srvr.h* and **_srvr.cc* files. (This conveniently reduces the number of files that need to be included in the server.)

Client-side stubs are not generated for the implementation of the concrete interface, since clients must program against the pure, base interfaces inherited by the concrete interface. As before, the client-side stubs for the base interface must be generated by using *contocc*, and the resulting **.h* and other files duly included by interested client code.

The dependencies can be illustrated as follows:



The following Makefile is used:

```
include VarRules
include $(OS)/Rules
include $(OS)/PT_Rules
```

```
include ContoccRules

ENV = messenger.env

ARITH_IDL_FILES = data.idl first_arith.idl second_arith.idl
ARITH_CFILES = $(ARITH_IDL_FILES:%.idl=%_srvr.cc)
ARITH_OFILES = $(ARITH_CFILES:%_srvr.cc=%_srvr.o)
ARITH_SHARED_CFILES = $(ARITH_IDL_FILES:%.idl=%.cc) \
        $(ARITH_IDL_FILES:%.idl=types_of_%.cc)
ARITH_SHARED_OFILES = $(ARITH_SHARED_CFILES:%.cc=%.o)

MSGR_IDL = messenger.idl
MSGR_CFILES = $(MSGR_IDL:%.idl=%_srvr.cc) messenger_impl.cc \
        message_server.cc
MSGR_OFILES = $(MSGR_CFILES:%.cc=%.o)
MSGR_SHARED_CFILES = $(MSGR_IDL:%.idl=%.cc) \
        $(MSGR_IDL:%.idl=types_of_%.cc)
MSGR_SHARED_OFILES = $(MSGR_SHARED_CFILES:%.cc=%.o)

MY_ARITH_SOURCES = my_arith.cc my_arith_server.cc my_arith_impl.cc
MY_ARITH_OFILES = $(MY_ARITH_SOURCES:%.cc=%.o)

PROGS = my_arith_server message_server client

all: $(ARITH_SHARED_CFILES) $(MSGR_SHARED_CFILES) $(PROGS)

my_arith.cc: $(ENV) my_arith.idl
    $(CONCRETE_STUBS) -e $(ENV) my_arith.idl

my_arith_server: $(ARITH_SHARED_OFILES) \
        $(ARITH_OFILES) $(MY_ARITH_OFILES)
    $(SDLD) -o my_arith_server $(ARITH_SHARED_OFILES) \
        $(ARITH_OFILES) $(MY_ARITH_OFILES) $(LIBRARIES)

message_server: $(ARITH_SHARED_OFILES) $(MSGR_SHARED_OFILES) \
        $(MSGR_OFILES)
    $(SDLD) -o message_server $(ARITH_SHARED_OFILES) \
        $(MSGR_SHARED_OFILES) $(MSGR_OFILES)  $(LIBRARIES)

client: client.o
    $(SDLD) -o client client.o $(MSGR_SHARED_OFILES) $(LIBRARIES)

clean:
    $(RM) $(PROGS) *.o core

superclean:
```

```
$(RM) $(PROGS) *.o core *.stats *.tm types_of_* \
    data.h first_arith.h second_arith.h messenger.h \
    my_arith.h *_stubs.h *srvr.* \
    data.cc first_arith.cc second_arith.cc my_arith.cc
```

See the *concrete_stubs* manpage for further information on command options.

Essential Spring Features 5

This chapter describes various essential features that Spring provides to programmers. These include elements of the Contract interface definition language, such as exceptions, enums, structures, unions, and sequences (arrays). Also included are Spring C++ interfaces for strings, clocks, random number generators, tables, queues, and threads.

This chapter also provides a description of Spring parameter passing modes, and of the equivalencies between primitive Contract types and C++ types.

5.1 Exceptions

Contract allows *exceptions* to be defined in interfaces. When called, the *contocc* compiler converts these to C++ exception code. An example of exception definition within an *idl* file can be found in the following location:

```
~tjh/spring_progs/chap5/exceptions
```

This two-domain program provides a *foo_factory* object via the name service. The factory's *create* method can be used to create a *foo* object, provided that the value required for initialization is specified as less than 100; otherwise, an exception is called.

The *foo.idl* file, whose interfaces are to be implemented by the server, is as follows:

```
module foo {

    exception too_high {long y;};
```

```
    interface foo {
        long get_value();
        void set_value(copy long new_value)
            raises (too_high);
    };

    interface factory {
        foo create (copy long a)
            raises (too_high);
    };
};
```

The exception is defined by the symbol *exception*, followed by the identifier for the exception, which in this case is *too_high*. A pair of curly braces follows, terminated by a semicolon. Optionally, within the curly braces, one or more parameters can appear, each terminated by a semicolon. Any Contract-defined parameter type can be used—for example, a string, a structure, an interface-type: here, a variable of Contract type *long* is specified (see the section *Type Equivalencies Between Contract and* C++, below).

Note – At the time of writing, a bug in the gcc compiler made it impossible to pass a Contract-defined object as an argument to an exception. The workaround is instead to pass a Contract-defined *structure* (see below), in one of whose fields the object is contained.

Further down within the module *foo*, two interfaces appear. Each contains an operation that raises the previously defined exception: as the final part of each operation-definition, the symbol *raises* appears, followed by the exception-name in parentheses.)

Thus, when the *contocc* compiler has been called on *foo.idl*, the following C++ exception-definition appears in the file *foo.h*:

```
exception {
    int y;
} foo_too_high;
```

The parameter defined for the exception, a Contract *long* type, has duly been converted to a C++ *int*. Any server-implementation that explicitly raises the *foo_too_high* exception (using the *raise* construct) can optionally specify an integer argument to the exception. The *foo_too_high* structure is then returned—containing the specified integer in its *y* field—to the invoker of the method, where the integer can be accessed.

Additionally within the *foo.h* file, the definition of the class *foo_factory* contains the following declaration of the *create* method:

```
class foo_factory
{
public:
    .
    .
foo_fp create(int arg_a) raises foo_foo_high, contract_fault;
    .
};
```

Proceeding from this, the server-implementation *foo_impl.h* header file represents the Contract-defined *create* operation as follows:

```
class foo_factory_impl:
    public foo_factory_srvr
{
public:
    virtual foo_fp create(int arg_a) raises foo_too_high;
};
```

And this method can be duly implemented in *foo_impl.cc* as follows:

```
foo_fp foo_factory_impl::create(int arg_a) raises foo_too_high
{
  if (arg_a >= 100) {
    raise foo_too_high(arg_a);    // Raise the exception.
  }

  foo_fp f;
  f = foo_fp (*new foo_impl() );
  f->set_value(arg_a);

  return f;
}
```

Thus, if the integer specified for initialization is too great, the exception *foo_too_high* is raised with the integer specified as its argument. The C++ structure containing the exception is then returned to the invoker of the method for appropriate handling. This occurs in *client.cc*:

```
  foo_factory_fp g = get_foo_factory();

  int x;
  cout << "Enter an integer below 100 for the value of \
          the foo object: ";
  cin >> x;
```

```
try {
  foo_fp alpha = g->create(x);
}
except ex {
  foo_too_high {
    printf("Exception foo_too_high was raised, \
          with value %d.\n", ex.y);
    delete &ex; // Delete the exception variable.
  }
  default {
    printf("The operation failed.\n");
  }
}
```

The call to the *foo_factory create* method is made within a *try* statement. This statement is followed by an *except* keyword, used to declare an exception variable, which is here named *ex* (the choice of name is arbitrary). This variable is itself followed by the name of the anticipated exception, and by directives appropriate for handling the exception.

When the *foo_too_high* exception is raised by the server, the variable *ex* is initialized to the value of the returned exception-structure. Thus, the field *y* becomes accessible to the client, and its value can be printed to standard output as part of an error message.

Note that the destructor for the exception variable must be called explicitly.

5.1.1 Raising and Handling Exceptions with Gcc

Within a single address-space, the gcc compiler allows C++-defined exception-structures to be passed by value, pointer, or reference. However, exceptions defined via Contract interfaces can only be passed by value.

5.2 Enums

An example of defining an *enum* in an *idl* file can be found in the following location, which provides another two-domain program:

```
~tjh/spring_progs/chap5/enums
```

A Contract enum definition is provided in *baz.idl*:

```
module baz {
```

```
    enum color {
        red, green, blue
    };
};
```

Thus, the syntax for the definition closely resembles that of a C++ enum definition. The definition is imported into *foo.idl*, along with a module from *bar.idl*, which provides a simple print operation:

```
import bar;
import baz;

module foo {

    exception whoops {long y;};

    interface foo : bar::bar {

        void junk_meth (consume baz::color x)
            raises (whoops);

    };
};
```

At this point, the appearance of the *foo.env* environment file should be noted:

```
$singleton: sc=singleton sc_file="subcontracts/singleton.h";
baz: file = "baz.idl";
bar: file = "bar.idl";
foo::foo: $singleton;
bar::bar: $singleton;
foo::whoops: $singleton;
```

Since it is imported into *foo.idl*, the module *baz* must indeed have its own *idl* filename and location specified within *foo.env*. However, *baz* does not require a subcontract definition, since the module contains no interfaces.

When called, the *contocc* compiler provides the following C enum definition, within the file *baz.h*:

```
enum baz_color {
    baz_color_red,
    baz_color_green,
    baz_color_blue
};
```

Once *client.cc* has obtained a *foo* object, it creates and passes *baz_color* enums as follows:

```
enum baz_color ruber = baz_color_red;
mm->junk_meth(ruber);

enum baz_color viridis = baz_color_green;
mm->junk_meth(viridis);

enum baz_color caeruleus = baz_color_blue;

try {
  mm->junk_meth(caeruleus);
} except another {
  foo_whoops {
    (void)printf("Exception was called, \
                  with value %d.\n", another.y);
  }
}
```

The *foo_impl.cc* definition of the method *junk_meth* is shown below:

```
void foo_impl::junk_meth(baz_color arg_x) raises foo_whoops
{
  cout << arg_x;
  cout << "\n";

  char an_array[4][10] = {
    "zeroth",
    "first",
    "second",
    "third"
    };

  printf("The current array-element is %s.\n", an_array[arg_x] );

  if (arg_x == baz_color_blue) {
    raise foo_whoops(4);
  }
}
```

5.3 Structures

The following directory contains an example of defininining *structures* in Contract and implementing them:

```
~tjh/spring_progs/chap5/structures
```

As in the case of enums, the definition of structures in Contract is highly similar to that in C++. This is indicated by the file *baz.idl*:

```
module baz {

    struct structure {
    long x;
    string s;
    };

};
```

The structure contains two fields, one containing a Contract *long* type (which will become a C++ *int*), the other containing a Contract *string*. The *string* will become a Spring-defined *string* object, defined in the header file *string.h*. (A description of strings is provided later in this chapter.)

Once again, the module *baz* is imported into *foo.idl*. The structure is used to specify the parameter of an operation:

```
import bar;
import baz;
module foo {

    interface foo : bar::bar {

        void junk_meth (consume baz::structure x);

    };

};
```

This leads *contocc* to provide the following C++ implementation in the file *baz.h* (subcontract methods have been omitted):

```
struct baz_structure
{
    int x;
    string s;
    baz_structure();
    baz_structure(int _x, string _s);
```

```
    baz_structure(baz_structure &_o);
    ~baz_structure();
    void operator=(baz_structure &_o);
        .
};
```

As well as providing the expected *int* and *string* fields, the definition also includes the overloaded = operator, allowing initialization of one structure from another, plus three constructors: the first creates without initialization of fields, the second creates *with* such initialization, the third creates with field-initializations identical to those of an existing structure.

The file *client.cc* creates a structure and passes it to the *junk_meth* method, as indicated below:

```
baz_structure my_struct;

my_struct.x = 0;
my_struct.s = string("leaving client\n");

cout << (my_struct.s).s2c();
cout << "\n";

mm->junk_meth(my_struct);
```

The server implementation file *foo_impl.cc* demonstrates how the structure can be used. For example, it uses the structure passed by the client as the basis for a further structure, created with the constructor and appropriate arguments:

```
baz_structure new_struct;
new_struct = baz_structure(200, "the new structure");
```

Additionally, it creates a new structure by referencing an existing one:

```
baz_structure recreated_struct;
recreated_struct = baz_structure(new_struct);
```

Finally, it creates a structure by assignment:

```
baz_structure assigned_struct;
assigned_struct = recreated_struct;
```

Note that the new structures created by these means are not shallow copies of the orginals. They are entirely separate entities, and their values can be modified independently.

5.3.1 Structures versus Objects

In most cases, objects (that is, as generated from *idl* interfaces or simply from C++ classes) are preferred to structures in Spring programming. A description of *queues* is provided later in this chapter: these permit objects to emulate the functionality of linked lists of structures.

5.4 Unions

Contract definition of a *union* is exemplified in the following location:

```
~tjh/spring_progs/chap5/unions
```

The file *baz.idl* contains the following Contract definition of a union:

```
module baz {

    union the_union
          switch (enum the_tag {the_int, the_bool, the_string, the_real}){
              case the_tag::the_int:
                  long the_int;
              case the_tag::the_bool:
                  boolean the_bool;
              case the_tag::the_string:
                  string the_string;
              default:
                  double the_real;
    };

};
```

As this indicates, the definition should commence with the symbol *union*, followed by an identifier. A *switch* statement and type-specifier follow. In this case, a Contract-defined *enum* is used. In fact, a *char, boolean, signed* or *unsigned short* or *long*, or appropriately scoped *name* can also be used (see below for additional examples). Finally, a series of *case* labels is provided, including a *default*.

Running the *contocc* compiler provides the following C++ definitions in *baz.h* (subcontract methods have been omitted from class *baz_the_union*):

```
enum baz_the_union_tag {
    baz_the_union_tag_NULL,
    baz_the_union_tag_the_int,
    baz_the_union_tag_the_bool,s
```

```
    baz_the_union_tag_the_string,
    baz_the_union_tag_the_real
};

class baz_the_union
{
public:
    baz_the_union() : tag(baz_the_union_tag_NULL) { }
    baz_the_union(baz_the_union &o);
    ~baz_the_union();
    baz_the_union &operator=(baz_the_union &o);
    operator int() { return tag; }
    baz_the_union_tag get_tag() { return tag; }
    bool is_NULL() { return (tag == baz_the_union_tag_NULL); }
   bool is_the_int() { return (tag == baz_the_union_tag_the_int); }
    bool is_the_bool() \
              { return (tag == baz_the_union_tag_the_bool); }
    bool is_the_string() { return (tag ==\
                          baz_the_union_tag_the_string); }
    bool is_the_real() { return \
                           (tag == baz_the_union_tag_the_real); }
    int &get_the_int() raises oneof_fault;
    bool &get_the_bool() raises oneof_fault;
    string &get_the_string() raises oneof_fault;
    double &get_the_real() raises oneof_fault;
    void set_the_int(int &the_int);
    void set_the_bool(bool &the_bool);
    void set_the_string(string &the_string);
    void set_the_real(double &the_real);
        .
        .
        .
private:
    baz_the_union_tag tag;
    union {
    int the_int;
    bool the_bool;
    char the_string[sizeof(string)];
    double the_real;
    } data;
    void unset();
};
```

The resulting class thus defines methods for getting and setting values for each of the union's possible types. The file *client.cc* creates and initializes a union as follows:

```
baz_the_union my_union;
my_union = baz_the_union();

int j;
j = 4;

my_union.set_the_int(j);
mm->junk_meth(my_union);

my_union.set_the_string("I'm a string.");
mm->junk_meth(my_union);
```

The server file *foo_impl.cc* implements *junk_meth* as follows:

```
void foo_impl::junk_meth(baz_the_union arg_x)
{
  if (arg_x.is_the_int() == 1) {
    cout << "The tag is " << arg_x.get_tag();
    cout << "\n\n";
    int x, y;
    y = 0;
    for (x = 0; x < 10; x++) {
      y = (y + arg_x.get_the_int());
      cout << y << "\n";
    }
  }
  else {
    cout << "The tag is " << arg_x.get_tag();
    cout << "\n\n";
    cout << (arg_x.get_the_string()).s2c();
    cout << "\n";
  }
}
```

5.4.1 Other Forms of Switch Type Specifier

As described above, a Contract *switch* statement accepts type specifiers of various kinds. An example of a Contract *boolean* used in this way is found in the following location:

```
~tjh/spring_progs/chap5/unions/other
```

Here, the appearance of *baz.idl* is as follows:

```
module baz {

    union the_union
```

```
    switch (boolean) {
        case 0:
           long the_int;
        case 1:
           string the_string;
    };

};
```

Still another example is found in the following directory:

```
~tjh/spring_progs/chap5/unions/another
```

The *baz.idl* union now features a *char* as a type specifier:

```
module baz {

    union the_union
    switch (char) {
        case 'a':
           long the_int;
        case 'b':
           string the_string;
    };

};
```

Neither these nor any of the other possible forms of type specifier have an effect on the resultant stubs files, such as *baz.h*: the C++ union definitions remain exactly as described above.

5.5 Sequences

A Contract-defined *sequence* is an object that is similar to an array. In Spring, sequences have been implemented in preference to arrays, since sequences have, by comparison, more extensive functionality. An example of how to use sequences is found in the following location:

```
~tjh/spring_progs/chap5/sequences
```

To define a sequence in an *idl* file, the Contract *typedef* statement must be used. This is demonstrated in the file *baz.idl*:

```
module baz {

    typedef sequence<long> array_i;
```

```
};
```

The symbol *typedef* is followed by the symbol *sequence* with an appended template, which contains the name of the Contract type of the elements of the sequence—in this case, the type is a *long*. The statement concludes with the chosen name for the C++ sequence class.

The *contocc* compiler produces the file *baz.h*, which duly contains a definition for the class *baz_array_i*. This class itself inherits from the class *array_descriptor*, which is located in the file */interfaces/spring/stub_suppoert/contract_runtime.h*. The sum total of methods and constructors thus available for a sequence is described below—*st* stands for the name of the C++ sequence class, and *t* stands for the element type. Thus, in the case of the sequence defined above in *baz.idl*, *st* represents the C++ class *baz_array_i*, and *t represents* the C++ type *int*. Examples are taken from *client.cc* and *foo_impl.cc*.

- *st()*
  Create an empty sequence, with flexible bounds, and with the lower bound initially set to 0. (Flexible bounds allow elements to be added to the upper and lower extremes of the sequence. Elements added beneath the initial lower extreme are indexed by negative integers.)

- *st(int lwb, int upb, bool flexible=TRUE)*
  Create a sequence with the specified lower and upper bounds. If *flexible* is set to *FALSE*, any subsequent attempt to alter the bounds of the sequence will raise *array_fault (af_flex)*. Thus:

```
baz_array_i *n;
int w;
printf("Enter an int, specifying the array's \
                        upper bound (lower is 0): ");
cin >> w;
n = new baz-array_i(0, w);
```

- *st(st &0)*
  Create a copy of an existing sequence. The elements will be shared by the old and new sequences. Thus:

```
baz_array_i *cp_n;
cp_n = new baz_array_i(*n);
```

- *st &operator=(st &o) raises array_fault*
  Copy the argument sequence by assignment. The elements of the sequence will be shared by the old and new sequences.

- *bool flexible() const*
  Return *TRUE* if the sequence has flexible bounds, and *FALSE* otherwise.

- *t elem(int index) raises array_fault*
  void set_elem(int index, t &new_elem) raises array_fault
  t &operator[] (int index) raises array_fault
  elem returns a copy of the selected item; *set_elem* updates the selected item; *operator[]* returns a reference to the selected item, so that it can be used on either the left or right of an assignment operator. *array-fault(af_bad_index)* will be called if the index is not within the current bounds of the sequence. Thus:

```
cout << "\nEnter index for element to be changed: ";
int index;
cin >> index;

int an_element;
an_element = (*n)[index];
```

- *int lwb() const*
  int upb() const
  int size() const
  Return the lower bound, upper bound, and number of elements in the sequence. Thus:

```
cout << "The lower bound is ";
cout << n->lwb();
cout << " and its value is ";
cout << n->elem(n->lwb());
cout << "\n";
```

- *void set_elems (int index, st *new_elems) raises array_fault*
  Assign all the elements of *new_elems* to successive positions starting at *index* in the destination sequence.

- *st *slice(int lwb, int upb) raises array_fault*
  Create a sequence whose elements are a subset of the original. The elements will be shared by the old and new sequences. Thus:

```
baz_array_i *j;
int r, s;

cout << "Enter a lower bound for new sequence: ";
cin >> r;
cout << "Enter an upper bound: ";
cin >> s;
```

```
j = n->slice(r, s);
v = j->size();          // v is a previously-defined integer
cout << "Length of the slice is ... " << v << "\n";
```

- *void addu(t &new_elem) raises array_fault*
  void addl(t &new_elem) raises array_fault
  Add an element to either the upper or lower end of the sequence, adjusting the bounds accordingly; more space will be allocated if needed, and the data copied across. *array_fault(af_flex)* will be raised if the sequence does not have flexible bounds, and *array-fault(af_precious)* will be raised if the sequence data is currently marked *precious* (see below).

- *t remu() raises array_fault*
  t reml() raises array_fault
  Removes one element from the upper or lower end of the sequence, adjusting the bounds accordingly. The element removed is returned as the result of the function. The exception conditions are the same as for *addu/addl*.

- *void reset(int new_lwb) raises array_fault*
  The sequence is shrunk to zero size, and the lower bound is set to the new value specified. All data elements are discarded. *array_fault(af_flex)* will be raised if the sequence does not have flexible bounds.

- *st(int lwb, int upb, t *imported_data)*
  *const t *export(bool precious=FALSE)*
  The constructor builds a sequence from the *imported_data*. The data is marked as *precious*, which will stop the sequence from being increased in size beyond the bounds given, and will stop the memory containing the data from being discarded. Thus:

  ```
  int [k] = {1, 2, 3, 4}
  baz_array_i *d;
  d = new baz_array_i(0, 3, k);
  ```

  export permits "back-door" access to the data of the sequence, and should be used with extreme caution. If the argument is *TRUE*, the data will be marked to prevent its deletion; if the argument is *FALSE*, no change in the *precious* state will be made. Thus:

  ```
  int new_k[4];
  const int *k_ptr;
  k_ptr = &new_k[0];
  k_ptr = d->export();
  ```

```
for (y = 0; y < 4; y++) {
    z = *k_ptr;
    k_ptr++;
    cout << "\t" << z << "\n";
}
```

- *bool precious()*
  void set_precious(bool precious)
  Return a bool indicating whether data is marked as *precious*; set the bool, thereby indicating whether or not the data should indeed be so marked.

- *~st()*
  The sequence is deleted, along with the data—provided that the data is neither shared nor marked *precious*.

5.6 Parameter Passing Modes

In conventional programming languages, such as C++, pass-by-value and pass-by-reference are two common parameter passing modes. Contract defines a set of parameter passing modes that are suitable both for objects and for primitive data types that are to move between client and server domains (which may in some cases be the same domain). Use of these modes is exemplified by the two-domain program in the following location:

```
~tjh/spring_progs/chap5/param_modes
```

The file *foo.idl* appears as follows:

```
module foo {
    interface bar {
        void print_msg();
        void set_a(consume long a);
        long get_a();
    };

    interface foo {
        void pass_copy (copy bar x);
        void pass_consume (consume bar y);
        void pass_borrow (borrow bar z);
    };
};
```

The module *foo* contains two interfaces, *foo* and *bar*. Each operation defined for *foo* is void and takes a bar as its argument. The *copy, consume,* and *borrow* parameter passing modes are all exemplified—the modes are described as follows:

5.6.1 Copy

This is equivalent to the pass-by-value mode of conventional programming languages. When the argument is an object, a copy of the object is created by means of the *standard copy operation* (the subcontract method provided in each C++ object-type class definition): the copy is passed to the invokee, while the invoker keeps the original. The exact implementation of the standard copy operation varies from one subcontract to another—the copies may or may not share state. When the *singleton* subcontract is used on an object, the copies do indeed share state, and any change made to one of the copies is immediately reflected in the other.

To demonstrate the *copy* mode, the file *client.cc* obtains a *foo_bar_fp* (using the function *get_foo_bar()*), and duly passes it as the argument to the *pass_copy* implementation method. When this has returned, the client prints to standard output the current private data-value of the copy of the object that it itself retained:

```
foo_bar_fp copy_fb;
copy_fb = get_foo_bar();
m->pass_copy(copy_fb);
int w = copy_fb->get_a();
cout << "After pass_copy operation returns, ";
cout << "client-side a-value is " << w << "\n\n";
```

In *foo_impl.cc*, the *pass_copy* method contains the following lines:

```
cout << "Server has the foo_bar_fp, passed copy\n";
arg_x->set_a(300);
int q = arg__x->get_a();
cout << "Server-side a-value is now set to " << q << "\n";
```

Thus, the data-values of the two copies are printed out in succession, respectively by server and client. This portion of the program produces the following standard output:

```
Server has the foo_bar_fp, passed copy
Server-side a-value is now set to 300
After pass_copy operation returns, client-side a-value is 300
```

This indicates that, as mentioned above, the *singleton* subcontract implements a *shallow copy* operation, whereby copied objects share the same state.

5.6.2 Consume

This is also equivalent to pass-by-value, except that it is illegal for the invoker to attempt to use the argument after invocation has occurred: the argument moves from the invoker domain to the invokee domain at the start of the operation, so that the invoker no longer has the argument (unless the invoker is also the invokee).

Thus, if the invoker wishes to access the argument (for example, an object) following invocation, the invoked method needs to be one that returns the argument. Alternatively, a form of invocation such as the following, shown in *client.cc*, could be used:

```
foo_bar_fp consume_fb;
consume_fb = get_foo_bar();
m->pass_consume(consume_fb->copy());
```

Here, the subcontract-defined *copy* method is itself called on the fat pointer: the resulting copy is thus passed as the consumed argument. And consequently, once again, changes made to that copy on the server side are also reflected in the copy of the object that remains on the client side.

5.6.3 Produce

This is the standard way whereby a result is returned. Again, it is equivalent to pass-by-value, except that it is illegal for the invokee to attempt to use the argument after returning: the argument moves from the invokee domain to the invoker domain, so that the invokee no longer has the object (unless it is also the invoker domain).

In the *foo.idl* file, the sole operation that returns a value is the *get_a* operation, defined within the *bar* interface. This value is returned in *produce* mode, as are all values returned in Spring: the keyword *produce* does not have to be specified.

An alternative use of the *produce* mode is for the specification of *out* parameters passed to an operation. The operation returns values into the specified parameters, which in the C++ implementation are passed by reference. An example of this can be found in the Spring clock interface, *clock.idl* (the clock is

discussed in greater detail later in this chapter). The clock's *get_fine_time* operation, which returns seconds and microseconds since boot time, is defined as follows:

```
void get_fine_time(produce long seconds, produce long micros);
```

In the C++ implementation header file, *clock.h*, the method is defined in this way:

```
void get_fine_time(int &arg_seconds, int &arg_micros) \
                   raises contract_fault;
```

5.6.4 Borrow

This is equivalent to pass-by-reference, except that it is illegal for the invoker to attempt to use the argument after invocation has occurred; the argument can be used again by the invoker after the operation has concluded (at which point it then becomes illegal for the invokee to attempt to use it).

The *pass_borrow* operation, defined within the *foo* interface, has the following definition within the *~tjh/spring_progs/chap5/param_modes/foo_impl.h* implementation file:

```
virtual void pass_borrow(foo_bar_fp &arg_z);
```

As this indicates, a Contract argument that is passed *borrow* is passed by reference in the C++ implementation. As would be expected, when the invocation has concluded, changes made to the argument by the invokee are reflected in the value maintained by the invoker.

5.7 Type Equivalencies Between Contract and C++

Contract provides various *primitive data types*, which are equivalent to C++ types. These equivalencies are shown in the following table:

Contract	C++
octet	unsigned char
boolean	bool
unsigned short	short unsigned int
unsigned long	unsigned int
unsigned long long	long long unsigned int
char	char
short	short int

long	int
long long	long long int
float	float
double	double
string	string

5.8 Strings

The Spring *string* is a C++ class type defined in the header file *util/string.h*. A two-domain code example showing how strings can be used resides in the following location:

`~tjh/spring_progs/chap5/strings`

Code exerpts shown below are from the files *client.cc* and *foo_impl.cc*, contained within this directory.

The public methods provided by the *string* class are as follows:

- *string()*
  Returns an empty string—the value "".

- *string(string &s)*
  Returns a string initialized from an existing one.

```
string other_string;
other_string = string(str);
```

- *string(const char *chars)*
  Returns a string made by copying the given C *chars*.

```
string new_str;
new_str = string("\nText for initializing string.\n");
```

- *string(const char *chars, unsigned int length)*
  Returns a string made by copying the contents of the buffer specified.

```
string str;
printf("Enter a string: ");
fgets(line, sizeof(line), stdin);
str = line;
```

- *~string()*
  The destructor.

- *void operator=(string &s)*
  Assign to string on left of = operator the value of string on right.

```
string second_str;
second_str = string(line, len);
string other_str;
other_str = second_str;
```

- *unsigned int length() const*
  Returns the number of characters in the string.

```
unsigned int b;
b = str.length();
```

- *unsigned int hash(unsigned int max) const*
  Returns the value from executing a hash function on the string. The result is in the range *0 <= value < max*.

- *Comparison operators:*
  bool operator==(const string &s) const
  bool operator!=(const string &s) const
  bool operator<(const string &s) const
  bool operator>(const string &s) const
  bool operator<=(const string &s) const
  bool operator>=(const string &s) const
  *bool operator==(const char *cp) const*

- *char operator[](unsigned int index) const;*
  Index operator, which returns a *char* rather than a reference (thus, it cannot be used for updates).

```
char c;
int k;
cout << "Enter a position-indicating integer: ";
cin >> k;
c = str[k];
```

- *Concatenation operators:*
  string operator+(const string &s) const
  string operator+(char ch) const
  Return a string that is a concatenation of the string on the left of the + operator and the input argument on the right.

```
string x, y, z;
x = "hello";
y = "goodbye";
z = x + y;
```

```
string z2;
z2 = z + "...this is a concatenated line";
```

- *const char *s2c() const*
  Converts a string to a *char**. If the client wants to change the *char**, they must themselves make a copy, then change the copy. This is necessary since other clients may share the same character vector.

```
cout << "New string is...";
cout << second_str.s2c() << "\n";
```

- *char *s2c_copy() const*
  Identical to *s2c*, except that the returned *char** is a copy of the body. The client is responsible for deallocating the returned *char**, using *delete*.

- *string substring(unsigned int pos, unsigned int length) const*
  Return a new string formed from the current string, starting at the character indicated by *pos*, and containing the number of characters indicated by *length*.

```
string j;
j = x.substring(1, 5); // Returns "ello" from "hello"
```

- *string rest(unsigned int pos) const*
  Return a new string, formed from the tail of the current string, starting at the character indicated by *pos*.

```
string e;
e = string(arg_x); // e becomes "experimental"
string ff;
ff = e.rest(4);    // ff becomes "rimental"
```

- *int find_char(char ch, unsigned int start = 0) const*
  Search the string for the given character and return the first-encountered position of that character, or -1 if there is no match. *start* is the character offset at which to start the search.

```
char chr = 'e';
int intgr;
intgr = arg_x.find_char(chr) // arg_x is "experimental"
cout << "Position of character is " << intgr << "\n"; // 0
```

- *int find_string(const string &s, unsigned int start = 0) const*
  Search for the given string *s* within the current string, return the position of the start of the match, or -1 if there is no match. *start* is the character offset at which to start the search.

```
intgr = arg_x.find_string("rim");
cout << "String's start-position is " << intgr << "\n";  // 4
```

5.9 Clocks

Spring provides a clock object. A program demonstrating its functionality is in the following location:

```
~tjh/spring_progs/chap5/clock
```

The Contract interface is itself defined in *interfaces/spring/dev/clock.idl*. To use the clock, the header file *dev/clock.h* should be included by the client.

The *idl* interface is as follows:

```
interface clock {
    void pause(copy long milliseconds) raises (alerted::alerted);
    time get_time();
    void get_fine_time(produce long seconds, produce long micros);
};
```

The clock object is obtained by a call to a static method in *spring_lib*. This library, which has been used earlier for the *become_server* call used to create servers, is discussed in greater detail later in this chapter. In *client.cc*, the call is made as follows:

```
clock_fp the_clock;
the_clock = spring_lib::get_clock();
```

The operations provided by the clock are examined below.

- *pause(copy long milliseconds)*
  Causes a pause for the specified number of milliseconds.

- *get_fine_time(produce long seconds, produce long micros)*
  Returns the number of seconds and microseconds that have elapsed since boot time. Note that the parameters are passed in *produce* mode, indicating that in the C++ implementation, they are *out* parameters, passed by reference. Thus, their C++ implementation definition appears as follows:

  ```
  void get_fine_time(int &arg_seconds, int &arg_micros) \
                    raises contract_fault;
  ```

  In *client.cc*, this method is called as follows:

  ```
  int secs, micros;
  the_clock->get_fine_time(secs, micros);
  cout << "Fine time returns " << secs << " seconds and ";
  ```

```
cout << micros << " microseconds.\n";
```

- *time get_time()*
  Returns a *time* structure, which contains fields representing the current time.The strucutre is itself defined within the *clock* module as follows:

```
struct time {
    // Microsecond precision is not guaranteed.
    long microsecond;
    long second;
    long minute;
    long hour;
    long weekday; // Sunday = 0
    long day; // 1 -> 31
    long month; // 0 -> 11
    long year; // e.g. 1990
};
```

  Use of the structure can be exemplified as follows:

```
struct clock_time the_time;
the_time = the_clock->get_time();
int x;
x = the_time.year;
cout << "The year is " << x << ".\n";
```

5.10 Random Numbers

A random number generator C++ class is provided from the include file *util/random.h*. This uses the following *linear congruential method* of random number generation, described in D.E.Knuth's *Seminumerical Algorithms* (*q.v.*):

$X[n + 1] = (X[n] * a + c) \bmod m$

Where $a = 1103515245$, $c = 12345$, and $m = 2^{31}$

This produces a series of approximately four billion numbers, which are used as the source for "random" number generation. A *seed* can be specified by the user of the generator—this determines the point within the series at which number-generation begins. Thus, if the series contains the subseries *8, 31, 39, 33*, then specification of *8* as the *seed* will determine that *31* and *39* are the first two random numbers to be returned (assuming that these numbers are within whatever *high-low* bounds have been specified by the user—if a *high* bound of *38* has been specified, then *39* is skipped over, and *33* is returned as the second random number).

The random number generator can also be used to generate a real number between *0* and *1*.

Use of a random number generator is demonstrated in the following location:

`~tjh/spring_progs/chap5/random`

Within the file *client.cc*, a random number generator is created with the *random()* constructor. This optionally takes an unsigned integer as an argument, to specify the *seed*. If no unsigned is specified, a *seed* is automatically generated from the system clock.

Once created, the *client.cc* generator is used to generate three random numbers, by means of the *next* method. This method can be called either with a single integer, specifying the upper bound of the range in which the number should lie, or with two integers, respectively specifying the lower and upper bounds:

```
random *a_gen;
a_gen = new random();
        .
int first = a_gen->next(HIGH);  // HIGH is defined above as 1000.
int first = a_gen->next(HIGH);
int first = a_gen->next(HIGH);
        .
```

When invoked with no *unsigned* argument, the *next* method returns a random *double* between *0* and *1*:

```
double d;
d = a_gen->next();
cout << d << "\n";
```

The *random* class provides a method, named *current*, that returns the object's current random number in the series. If this number is itself specified as the *seed* of a new object of this class, both objects have the same state, and the invocation of *next* on each object in turn returns the same random number (whether integer or double) in both cases. Thus:

```
unsigned the_current = a_gen->current();
random *another_gen = new random(the_current);

int latest_int = another_gen->next(HIGH);
printf("\tNext int from new generator is %d\n", latest_int);

int old_int = a_gen->next(HIGH);
printf("\tNext int from previous generator is also %d\n", old_int);
```

The integers respectively returned by the invocations of *next* on the two generators are indeed the same. (This works for floats as well as integers.)

The *random* class also provides a method named *seed*, which duly returns the seed according to which the current object was created. If this *seed* is itself specified as the *seed* of a new object of this class, the random number series of the new object will be identical to that of the old, allowing the progression through the old series of numbers to be re-enacted by invocations of *next* on the new object. Thus:

```
unsigned returned_seed = a_gen->seed();
random *a_new_gen = new random(returned_seed);
int new_first = a_new_gen->next(HIGH);
int new_second = a_new_gen->next(HIGH);
int new_third = a_new_gen->next(HIGH);

printf("\tFirst three rands from a_new_gen are %d, %d, %d.\n", \
       new_first, new_second, new_third);
```

This code returns integers identical to those returned by the invocations of *next* on *a_gen*, which appear earlier in *client.cc*, and were shown previously in this section.

5.10.1 Methods on Random Number Generators

- *random(unsigned seed = 0)*
  Returns a random number generator. If the seed is 0, a random seed from the clock is generated.

  ```
  random *a_gen;
  a_gen = new random();
  ```

- *int next(int high)*
  Returns a random integer in the range 0 to *high*.

  ```
  int first = a_gen-next(HIGH);
  ```

- *int next(int low, int high)*
  Returns the next random integer in the region *low* to *high*.

- *double next()*
  Returns the next number uniformly distributed between 0 and 1.

  ```
  double d;
  d = a_gen->next();
  ```

- *unsigned seed()*
  Returns the seed. This can be used to create a new random number generator that is based on the same random series as that of the old.

```
unsigned returned_seed = a_gen->seed();
```

- *unsigned current()*
  Returns the current value of *X[n]*, which is the point in the series at which the generator is currently situated. If this value is specified as a seed, a new random number generator with the same state as that of the old can be created.

```
unsigned returned_seed = a_gen->seed();
random *a_new_gen = new random(returned_seed);
```

5.11 Tables

The Spring header file *interfaces/spring/util/table.h* provides a class for the definition of C++ hash table objects. It also provides a number of standard hash and equals functions, including ones for integers and for strings. Programs demonstrating this functionality are in the following locations:

```
~tjh/spring_progs/chap5/int_fun
~tjh/spring_progs/chap5/string_star_fun
~tjh/spring_progs/chap5/both_fun
```

The *table.h* class *table* provides the following constructor for a hash table:

```
table(int initial_size, hash_function *hasher, equal_function \
                                                *equals)
```

The *table_size* parameter specifies the number of *buckets* into which the hash table is partitioned: each bucket can hold one record. The other two parameters respectively specify pointers to defined functions, both of which are *typedef*'d by *table.h*, as follows:

```
typedef int hash_function(KEY k, int max);
typedef bool equal_function(KEY a, KEY b);
```

Elsewhere in *table.h*, the *KEY* is itself *typedef*'d to be an integer. Thus, both functions take two integers as their arguments. In the *hash_function*, *KEY k* describes the integer-identifier of a record to be stored in a hash table, while *int max* describes the maximum number of buckets that the table will contain. The function assigns the record to one of the buckets, based on the result of an

arithmetical computation. The exact nature of this computation can be determined by the table's user—a function can be defined to perform the computation, and the function name declared as a *hash_function*.

In the *equal_function*, the two *KEY* arguments describe two hash table keys. The function, which can likewise be defined and declared by the user, returns a *bool*, to determine whether the keys are duplicates. No duplicate keys are permitted within the hash table—attempts to insert duplicates will result in a panic. The *equal_function* can thus be used to diagnose potential collisions, before record-insertion is attempted.

The *table.h* file provides standard *hash* and *equal* functions for use with integers and strings, and declares these functions as follows:

```
hash_function hash_int;
equal_function equal_int;

hash_function hash_string_star;
equal_function equal_string_star;
```

In the example file *int_fun/client.cc*, a pointer to an object of type *table* is defined. The constructor is called with a size of *SIZE* (which is defined as 3), and with the standard *table.h*-provided hash and equal functions for integers:

```
table *t = new table(size, hash_int, equal_int);
```

Objects of user-defined type *my_class* are then created. Each has an integer and a string as private data, and these values are instantiated by means of the index value of a *for* statement, and the contents of a two-dimensional character array. Once the objects have been created, they are inserted in the table by means of the *table.h*-defined *insert* method, whose arguments are the unique *key* used to allocate the record to a bucket within the table, and the *value* (or *record*) to be inserted. Both arguments must be integers, and casting is thus used to specify the *my_class* object to be inserted:

```
for (int i = 0; i < COUNT; i++) {
    my_class *the_class = new my_class(i, string_array[i]);
            .
    t->insert( the_class->get_a(), (int)the_class );
            .
```

The *insert* method uses the *hash_int* and *equal_int* functions, previously specified in the table's constructor, to perform insertion of records into the table. (In fact, the standard *hash_int* function uses the *modulo* operation to determine the destination bucket: thus, for *key* 7 and table-size 3, the hash

function allocates the record to bucket 1.) If the *equal_int* function returns *TRUE*, a panic occurs—*client.cc* proceeds under the assumption that there will be no collision, and does not perform checks.

Subsequently in this file, the *first_key* and *next_key* methods are used, to iterate through the contents of the table, returning all the entered keys:

```
for (i = 0; i < COUNT; I++) {

    if (i == 0) {
        the_key = t->first_key();
    }
    else {
        the_key = t->next_key(the_key);
    }

    printf("\t%d\n", the_key);

}
```

The method *next_key* takes as its argument the previous key. The method panics if an unknown key is passed.

The *lookup* method can be used to return an integer corresponding to a value entered in the table—a cast operation can then be performed to return the value as an appropriate type. The *lookup* method takes as its argument an integer that corresponds to the *key* under which the value was entered:

```
value = t->lookup(num);
my_class *returned_class = (my_class *)value;

string returned_string = returned_class->get_str();
```

Since integers must be used to specify both keys, casting is used more extensively when the keys are to be determined by different type—for example, strings. In the example file *string_star_fun/client.cc*, the *insert* method is called with an integer key that is derived from a pointer to a string:

```
my_t->insert( (int)the_class->get_str_star(), (int)the_class );
```

Following retrieval by means of iteration, each key is duly cast back to its string value, to be printed to standard output:

```
string *zstring = (string *)the_key;
printf(\t%s\n", zstring->s2c() );
```

To perform lookup, C *chars* are taken from standard input, and, minus the return character, are used to initialize a Spring string object. A pointer to this object is then cast as an integer, and is passed as the argument to the *lookup* method. Finally, the integer returned by *lookup* is itself cast as the appropriate type, *my_class*:

```
char line[10];
string *in_str;
int value;
my_class *returned_class;
int returned_int;

printf("Now enter a key (a string) for looking up an object: ");
fgets(line, sizeof(line), stdin);
line[strlen(line)-1] = '\0';
in_str = &(string(line));

value = my_t->lookup( (int)in_str );
returned_class = (my_class *) value;
returned_int = returned_class->get_a();
printf("Integer value of chosen object is \"%d.\"\n", returned_int);
```

The example file *both_fun/client.cc* can be inspected for a demonstration of generating a key from both the string and the integer values of an object, and of checking for duplicates before calling *insert*.

5.11.1 Contents of table.h

The *table.h* include file contains the following elements, intended for general programming use:

5.11.1.1 Typedefs

- *typedef int KEY*
  typedef int T
  A key and its associated value. A value is stored in a table according to the unique key associated with it—a defined *hash_function* (see below) is performed, using the key and the maximum size of the table as arguments, and allocates associated value to a given bucket within the table. Since the key and value must be specified as integers, casting should be performed when different types are involved.

- *typedef int hash_function(KEY k, int max)*
  typedef bool equal_function(KEY a, KEY b);
  The *hash_function* takes two arguments: *KEY*, which is a unique integer associated with a value to be stored in the table; and *max*, which is the number of buckets in the table. The function assigns the record associated with *KEY* to one of the buckets, based on the result of an arithmetical computation. The exact nature of this computation can be determined by the table's user—a function can be defined to perform the computation, and the function name declared as a *hash_function*.

  The *equal_function* takes two KEY arguments, which describe two hash table keys. The function, which, like the *hash_function*, can be defined and declared by the user, returns a *bool*, to determine whether the keys are duplicates. No duplicate keys are permitted within the hash table—the method *insert*, which is indeed used to insert values into the table, makes an implicit call to the *equal_function*, and panics if *TRUE* is returned. The *equal_function* might be used independently of *insert*, to diagnose potential collisions before record-insertion is attempted.

5.11.1.2 Methods on Class table

- *table(int initial_size, hash_function *hasher, equal_function *equals)*
  The constructor for *table* takes three arguments: *initial_size*, which is an integer describing the maximum number of buckets in the table; **hasher*, which is a declared *hash_function*, and **equals*, which is a declared *equal* function. For example, a *hash_function* can be declared with the following statement:

  ```
  hash_function hash_function_name;
  ```

- *void insert(KEY key, T value);*
  Inserts the specified *value* into the hash table, provided that the *key* is unique—that is, it has not previously been used as the *key* of another inserted value. Both *KEY* and *T* are *typedef'd* within *table.h* as integers: therefore, other types that need to be used for keys and values should be cast to integers before being specified as arguments to *insert*.

- *void remove (KEY key)*
  Removes from the table the value associated with the specified *key*. Specifying a *key* for a value not in the table has no effect.

- *T lookup(KEY key)*
  Returns the value in the table that corresponds to the specified key, if one exists. Specifying a *key* for a value not in the table returns *NULL*.
- *KEY lookup_key(KEY key)*
  Returns the specified *key*, if the *key* is associated with a value currently in the table. Otherwise, returns *NULL*.
- *T remove_first_entry()*
  Removes and returns the first value in the table.
- *KEY first_key()*
  KEY next_key(KEY old_key)
  While *first_key* returns the *key* of the first value in the table, *next_key* takes as its argument the *key* most recently returned by *first_key* or *next_key*, and returns the *key* of the next value in the table. These methods can be used to iterate over all the elements in the table. Both return *NULL* if the end of the table has been reached; *next_key* panics if it is passed an unknown key.

5.11.1.3 Hash and Equals Functions

- *hash_function hash_int*
  equal_function equal_int
  Standard hash and equal functions for use with integers, which can be specified as the arguments to the *table* constructor.
- *hash_function hash_string_star*
  equal_function equal_string_star
  Standard hash and equal functions for use with strings, which can be specified as the arguments to the *table* constructor.

5.12 Queues

Spring provides C++ classes that allow creation and use of linked lists. These classes, defined in the header file *interfaces/spring/util/queue.h*, are as follows:

- *Queue*—the linked list itself
- *Qelem*—a class that must be inherited by the class intended to be part of a linked list
- *QueueIter*—an iterator over a queue

In this section, some observations are made regarding the possible use of these classes; then follows a full account of the methods of each class. A code example, cited throughout this section, is found in the following location:

```
~tjh/spring_progs/chap5/queues
```

5.12.1 Creating a Queue Element

A queue element is the instantiation of a client-defined C++ class that inherits from the Spring-defined class *Qelem*. Thus, *client.cc* contains the following:

```
class the_class : public Qelem {
    int x;
public:
    the_class (int a) {x = a;}
    void set_x (int b);
    int get_x ();
};
```

5.12.2 Creating a Queue

A *queue* is a linked list of objects. To create a queue, declare a queue variable; then initialize it with a single queue-element, which will be the head of the linked list—this initialization is performed with class Queue's *add-head* method.

In its private data, Class Qelem has two pointers—one to a Qelem that follows it, the other to a Qelem that precedes it. These elements can be created with with Qelem's *insert_after_this* and *insert_before_this* methods, and accessed with its *get_next* and *get_prev* methods.

Thus, the following code creates a queue, and then populates it with a linked list of queue elements:

```
the_class *a_class;
int i, value;

Queue *q;
q = new Queue();

for (i = 0; i < 5; i++) {
    if (i == 0) {
        a_class = new the_class(i);
        q->add_head(a_class);
    }
```

```
    else {
        a_class->insert_after_this(new the_class(i));
        a_class = (the_class *)a_class->get_next();
    }
}
```

Note that the values returned by *get_next* and *get_prev* are of type *Qelem*, and must therefore be typecast with *(the_class *)*.

The *head* (first element) and *tail* of a queue point to each other. Thus, invoking *get_prev* on the first element returns the last, while invoking *get_next* on the last returns the first. When the queue contains a single element, that element is self-looped. These facts are demonstrated in *client.cc*.

5.12.3 Creating and Using a Queue Iterator

A *queue iterator* provides a means of performing iterative operations on the elements of a queue. Its principal methods are *first* and *next*, which respectively return pointers to the first and next elements in a given queue.

The following code creates a queue iterator and demonstrates its principal methods:

```
QueueIter *qi;
qi = new QueueIter();

some_class = (the_class *)qi->first(q);
value = some_class->get_x();
printf("%d.\n", value);
printf("...which is the head of the queue.\n");

while (1) {
    some_class = (the_class *)qi->next();
    value = some_class->get_x();
    printf("\t%d\n", value);

    if (some_class == (the_class *)q->tail())
        break;
    }
    printf("...which is the remainder of the queue.\n");
```

5.12.4 Methods on Queues, Elements, and Iterators

This subsection lists the public methods provided by the objects *Queue*, *Qelem*, and *QueueIter*.

5.12.4.1 Methods on Queue

- *Queue()*
  Returns an empty queue. Thus:

```
Queue *q;
q = new Queue();
```

- *boolean empty()*
  Returns *TRUE* if the queue has no elements, FALSE otherwise. Thus:

```
boolean a_bln;
a_bln = q->empty();
```

- *void remove(Qelem *qe)*
  Removes the specified queue element from the queue. If the queue contains more than one element, the elements preceding and following this one are linked together. Thus:

```
some_class = (the_class *)qi->first(q);
while (1) {
    value = some_class->get_x();

    if (value == 3) {
        q->remove(some_class);
        break;
    }

    some_class = (the_class *)qi->next();
}
```

- *Qelem* head()*
  Returns the *Qelem* that is the head of the queue. Thus:

```
a_class = (the_class *)q->head();
```

- *Qelem* tail()*
  Returns the *Qelem* that is the tail of the queue. If the queue has only one element, that element is returned. Thus:

```
a_class = (the_class *)q->tail();
```

- *void add_head(Qelem *qe)*
  *void add_tail(Qelem *qe)*
  Add a new element to the start/end of the queue: the element is thus the new *head/tail*. Thus:

```
a_class = new the_class(i);
q->add_head(a_class);
```

- *Qelem* remove_head()*
  Qelem remove_tail()*
  Remove the element that is the *head/tail* of the queue. Thus:

```
new_tail_class = (the_class *)q->remove_tail();
```

5.12.4.2 Methods on Qelem

- *Qelem* get_next()*
  Qelem get_prev()*
  Return the next/previous element in the queue. If this element is the sole element, it is itself returned.

```
a_class = (the_class *)a_class->get_next();
```

- *void insert_before_this(Qelem *qe)*
  *void insert_after_this(Qelem *qe)*
  Inserts the specified new element before/after this element.

```
a_class->insert_after_this(new the_class(i));
```

- *void unlink()*
  Unlinks this element from the preceding and following elements in the queue. The preceding and following elements are then linked together, and this element itself is thus removed from the queue. (This method is equivalent to the *remove* method on class *Queue*.) Thus:

```
some_class = (the_class *)qi->first(q);
while (1) {
    value = some_class->get_x();

    if (value == 110) {
        some_class->unlink();
        break;
    }

    some_class = (the_class *)qi->next();
}
```

- *void selfloop()*
  Makes this element self-looped. Thus, its *get_prev* and *get_next* methods return itself. (See next bullet for example.)

- *boolean is_selflooped()*
  Returns *TRUE* if the element is selflooped, *FALSE* otherwise. Thus:

```
boolean a_bln;
a_bln = new_tail_class->is_selflooped();
if (a_bln == TRUE) {
    printf("\nThe tail has been removed, \
                            and is self-looped.\n")
}
else {
    new_tail_class->selfloop();
}
```

5.12.5 Methods on QueueIter

- *QueueIter()*
  Returns a queue iterator. Thus:

```
QueueIter *qi;
qi = new QueueIter();
```

- *Qelem *first(Queue *q)*
  Returns the queue element that is the head of the specified queue. Thus:

```
some_class = (the_class *)qi->first(q);
```

- *Qelem *next()*
  Returns the queue element that is next in the queue. This method is invoked subsequent to a prior invocation of *first* on a specific queue, and to zero or more prior invocations of *next*. Thus:

```
while (1) {
    some_class = (the_class *)qi->next();
    value = some_class->get_x();
    printf("\t%d\n", value);

    if (some_class == (the_class *)q->tail())
        break;
}
```

5.13 The spring_lib Utilities

The file *spring_lib.h* is a standard include file for Spring applications. It itself includes a variety of files, amongst which are the following:

```
#include "util/basic.h"
#include "util/thread.h"
#include "util/mutex.h"
#include "util/condition.h"
#include "dev/clock.h"
```

The file contains a class named *spring_lib*, which provides a number of static methods. The most generally important are the following:

- *static void become_server()*
  This method, which has been demonstrated throughout the previous chapters of this document, causes the program to become a server. The method does not return.
- *static clock_fp get_clock()*
  Returns a clock object for the current machine.

5.14 Definitions in basic.h

The file *basic.h* contains basic Spring definitions. (As indicated above, *basic.h* is included by *spring_lib.h*.) The file is as follows:

```
typedef int bool;
typedef bool boolean;
#define TRUE 1
#define FALSE 0
#define true TRUE
#define false FALSE

#define NULL 0

typedef unsigned int   u_int;
typedef unsigned short u_short;
typedef unsigned long  u_long;
typedef unsigned char  u_char;

typedef char *caddr_t;

#include "util/string.h"
#include "util/debug.h"
```

```
class thread;

typedef class domain_domain_1_0_fp domain_fp;
extern class domain_fp this_domain;

// "failure" is a standard exception for use in C++ libraries.
exception {string message;} failure;
```

5.15 Threads

Spring provides C++ objects for threads, mutexes, and conditions. This section descibes how these can be used in multi-threaded programming. A simple multi-threaded program can be found in the following location:

```
~tjh/spring_progs/chap5/threads
```

5.15.1 Defining and Running Threads

The *thread* C++ class is defined in the header file *interfaces/spring/util/thread.h.* (Note that it is not this file, but *util/spring_lib.h* that should be included in client **.cc* files.) This class should be inherited by all client-defined classes that wish to become threads. The *thread* class contains a virtual method named *startup,* which is invoked when a thread is run. The client-defined thread should redefine the *startup* method to perform whatever activities the client desires from its thread. Thus:

```
class first_thread : public thread {
public:
    void startup();
};
```

No constructor is needed.

Within the main program, after the client-thread has been created, the *thread* class' *make_runnable* method should be invoked. This in turn causes the client-defined *startup* method to be invoked.

```
first_thread *the_first_thread;
the_first_thread = new first_thread();

the_first_thread->make_runnable();
```

5.15.2 Using Mutexes and Conditions

Spring uses *mutexes* and *conditions* to schedule the running of threads. A mutex, whose C++ class is defined in *util/mutex.h*, is often declared globally within a client program: thus, each thread within the program can use it. The mutex class provides two principal methods: *lock* and *unlock*. When a thread calls *lock* on a mutex that is indeed available, the thread is granted possession of the mutex, and is able to continue execution: no other thread can possess the mutex until this thread releases the mutex.

If a thread calls *lock* on a mutex that has already been locked by another thread and has not yet been released, execution of the calling thread is suspended (that is, the thread *sleeps*). The suspended thread cannot resume execution until the mutex is released by its current owner.

If multiple threads have simultaneously called *lock* on an unavailable mutex, there is no predicting which will successfully obtain the mutex when it finally becomes available. However, some degree of control over this situation can be effected by use of *conditions*. The C++ class for a condition, defined in *util/condition.h*, provides *wait* and *notify* methods. If each of a number of threads has a condition associated with it, each thread can *wait* on its own condition (specifying the mutex it requires), and can *notify* on another thread's condition—a *notify* being performed when the notifying thread itself has a mutex that it is prepared to release. In this way, conditions create channels of communication, whereby threads conspire to pass a single mutex between them. (Success in the attempt is still not absolutely guaranteed.)

In *client.cc*, a mutex and three conditions are defined globally. The mutex is used by each of three user-defined threads, *first_thread*, *second_thread*, and *third_thread*. The conditions, *c1*, *c2*, and *c3*, are each associated with one of these threads. Additionally, *main* is considered to be the zero thread, and is associated with condition *c0*.The overall structure of the program is as follows:

```
mutex *the_mutex;
int count;
condition *c0 // This is main's own condition.
condition *c1;
condition *c2;
condition *c3;
    .
    .          // The three thread classes and their startup
    .          // methods are defined here.
    .
```

```
int
main()
{
    the_mutex = new mutex;

    c0 = new condition;
    c1 = new condition;
    c2 = new condition;
    c3 = new condition;

    count = 0;

    first_thread *the_first_thread;
    the_first_thread = new first_thread
        .
        .       // The other two threads are created here.
        .
    the_mutex->lock();      // main grabs the mutex first

    the_first_thread->make_runnable();
    the_second_thread->make_runnable();
    the_third_thread->make_runnable();

    c0->wait(the_mutex);

    the_mutex->unlock();

    return 0;
}
```

When all variables have been initialized, *main* invokes *lock* on the mutex. It then invokes *make_runnable* on each of the defined threads. Then, it calls *wait* on condition *c0*: this means that it releases the mutex and goes to sleep; it will remain asleep until it a *notify* call on *c0* is made by some other thread.

Now that the mutex has been released, the other threads are at liberty to lock it. The thread most likely (though not certain) to do this first is *first_thread,* since its *make_runnable* method was the first to be invoked by *main.* The *startup* method for *first_thread* is defined as follows:

```
class first_thread::startup() {

    the_mutex->lock();

    while (count < 10) {
```

```
            printf("first_thread...\n");

            count++;

            c2->notify();        // Prepare second_thread for mutex.
            c1->wait(the_mutex); // Release mutex, await its return.
          }

        c0->notify              // When count > 10, let main know, and
        the_mutex->unlock();     // then give up the mutex.

        printf("\nFirst thread done...count is %d\n", var_one);

}
```

Provided that *count* is not yet set to 10, *first_thread* begins by attempting to lock the mutex, and fails, since no *main* has already locked it. Thus, *first_thread* goes to sleep. Subsequently, when *main* releases the mutex, *first_thread* is probably the user-defined thread that wakes and successfully performs its *lock*. *first_thread* then enters a *while* loop, printing to standard output, and incrementing *count*. Next, it calls the *notify* method of the condition *c2*—*second_thread* is destined to wait on this condition, but since *second_thread* has not yet started, this call to *notify* actually has no effect. *first_thread* then calls *wait* on *c1*, which is the condition it will now continue to use as its own. In this *wait* call, it specifies *the_mutex* as its argument, thus indicating that *the_mutex* should be released, that *first_thread* itself is going to sleep, and that the return of *the_mutex* is the thing that will wake *first_thread* up.

second_thread, when made runnable by the main program, behave similarly to *first_thread*. Initially, it attempts to lock the mutex, but since the mutex is already held by *main*, it goes to sleep. When wakened by *first_thread*'s release of the mutex, *second_thread* prints to standard output, notifies on *c3*, and waits on its own condition, *c2*:

```
void second_thread::startup() {

    the_mutex->lock();

    while (1) {

        printf("\tsecond_thread...\n");

        if (count >= 10) {
            c1->notify();
            the_mutex->unlock();
```

```
            break;
        }
        else {
            c3->notify();        // Pass control to third_thread.
            c2->wait(the_mutex); // Wait for third_thread.

            printf("\tsecond_thread...\n");
            c1->notify();     // Wake up, pass mutex to first_thread.
            c2->wait(the_mutex);  // Wait for first_thread
        }
    }
}
```

After *second_thread* has invoked *wait* on *c2* and has gone to sleep, *third_thread* is made runnable by the main program:

```
void third_thread::startup() {

    the_mutex->lock();

    while (1) {

        printf("\tthird_thread...\n");

        if (count >= 10) {

            c2->notify();
            the_mutex->unlock();
            break;
        }
        else {
            c2->notify();
            c3->wait(the_mutex);
        }
    }
}
```

Thus, *third_thread* calls *notify on c2*, which is the condition that *second_thread* is waiting on. This call prepares *second_thread* for the return of the appropriate mutex. When *third_thread* subsequently releases *the_mutex*, *second_thread* is attentive to this, and is able to wake up. *second_thread*'s action at this point is to print (again) to standard output, notify *first_thread*, and release the mutex. Thus, it becomes *first_thread's* turn to get the mutex back, and resume its own execution.

When *first_thread* next does its *notify*—on *c2*—*second_thread* is indeed waiting, and is able to wake up and continue.

For each iteration, *first_thread* increments *count*. When *count* reaches 10, this is the signal for each of the threads to notify the next appropriate thread without waiting, invoked *unlock* on *the_mutex*, and end. Additionally, *first_thread* performs a *notify* on *c0*, allowing *main* to wake up, resume possession of the mutex, *unlock* the mutex, and return.

A sample of the output is as follows:

```
first_thread...
        second_thread...
                third_thread...
        second_thread...
first_thread...
        second_thread...
                third_thread...
        second_thread...
first_thread...
        .
        .
        .
```

5.15.3 Methods on Threads

- *virtual void startup()*
  Invoked when a thread is executed. This virtual method should be appropriately coded for each user-defined subclass of *thread*. Each subclass should then be instantiated, and the *make_runnable* method invoked—this in turn invokes the *startup* method.
- *make_runnable()*
  Invoked on a newly constructed thread object to make it runnable—the user-coded *startup* method is invoked.
- *bool is_alerted()*
  Returns *TRUE* if a thread has an *alert* outstanding, *FALSE* otherwise.
- *void clear_alert()*
  Clears any alert that is outstanding.
- *static thread *current()*
  Returns the C++ thread object for the current thread.

5.15.4 Methods on Conditions

- *condition()*
  Returns a condition.
- *void wait(class mutex *m)*
  Releases the specified mutex and suspends execution of the calling thread. The mutex is *atomically* freed; *i.e.* no *notify* or *broadcast* can intervene between the calling thread's freeing of the mutex and its suspending of execution. When the mutex is made available again—by a *notify* on this condition, or by a *broadcast*—it is *non-atomically* reclaimed by the calling thread.
- *void alertable_wait(class mutex *m) raises alerted;*
  Similar to *wait*. However, if an *alert* is sent to the waiting thread, the thread resumes execution, even though no *notify* or *broadcast* has occurred. If an *alert* is sent while the thread is not waiting, or if the thread is already alerted when it calls *alertable_wait*, the *alerted* exception is raised.
- *void notify()*
  Wakes up one or more threads that are waiting on this condition. If multiple threads are waiting on the condition, *notify* only guarantees to wake up an arbitrarily chosen one, though it may also wake up others.
- *void broadcast()*
  Similar to *notify*. However, *broadcast* guarantees to wake up all threads that were sleeping on this condition at invocation.
- *void wait_without_mutex()*
  void alertable_wait_without_mutex()
  Similar to *wait* and *alertable_wait*, except that no mutex is involved. Considerable care should be exercised when using these methods.

5.15.5 Methods on Mutexes

- *mutex()*
  Returns a mutex.
- *void lock()*
  Attempts to lock the mutex on which the method is invoked. If the mutex is not currently free, *lock* suspends until the mutex is indeed made free.
- *void unlock()*
  Releases the mutex and wakes up a waiting thread (if one exists).

- *int must_be_held()*
  int must_not_be_held()
  Debugging assertions. The mutex either must or must not be locked by the current thread—otherwise, a panic occurs.

- *bool try_lock()*
  If the mutex is currently free, *try_lock* claims it and returns *TRUE*, otherwise returns *FALSE*.

- *void become_owner()*
  Bypasses normal ownership assertions when unlocking a mutex. This is intended only for use in exceptional circumstances, such as program startup.

The Name Service 6

The Spring Name Service allows client programs to bind objects to names, and resolve previously bound names to objects. Any name can be associated with any object; thus, the system permits a uniformity that is not possible in a system such as UNIX.

A principal element in the organization of the Name Service is the *context*, which is an object that contains defined name-to-object associations, known as *name bindings*. Thus, a context is roughly analogous to a directory in UNIX , while the names within a context are analogous to files or subdirectories.

A *compound name* is an object-name that contains several components, indicating the object's relation to a context within which it is nested at some level: the compound name thus may have the appearance of a UNIX pathname. A *simple name* is a name with a single component. The total naming area spanned by a given context is known as a *name space*.

Although comparisons with the UNIX filesystem are useful, it is again important to note that the Spring name service is far more inclusive. Whereas the UNIX filesystem does indeed only provide naming for files—the naming for other entities, such as processes or environment variables, being handled differently—the Spring name service provides a single model for the naming of *all* objects, whatever they represent.

Frequently, when a server process implements an object that it wishes to make available for the use of unspecified clients, it registers the object with the name service, specifying a name for the object, and the context in which it wishes the object to reside: this is known as *binding*. The choice of name is entirely

arbitrary, and does not need to correspond to the name of the object's interface. When a client wishes to access a published object, it must reference the appropriate context and object name: this is known as *resolving*. Use of the name service, then, represents the most typically used object exchange mechanism for Spring programs.

This chapter describes various aspects of the Name Service, including how to bind, resolve, and unbind contexts and other objects. It explains how to gather information on existing bindings, how to create symbolic links, and how to use access control lists.

The chapter also describes the interfaces for *freezing* and *melting*, whereby an object can be conserved and passed around by means of an associated token and a server (or *object-manger*) identifier. Though these interfaces may not frequently be needed by the client programmer, they provide insight into one of the basic internal mechanisms whereby the Name Service stores and retrieves application-defined objects.

6.1 Binding Objects

Spring provides two main methods for binding objects to names: *bind* and *rebind*. The *bind* method only performs the bind if the specified binding does not already exist; if the binding does in fact exist, the *rebind* method can be used to overwrite it.

The methods for *bind* and *rebind*, found in *naming/naming.h*, are static methods on the class *naming*. This class is declared with the use of a C++ *template*:

```
template <class T>
class naming<T>
{
    .
    .
```

The template permits a class to be specified in each invocation of *naming's* static methods. The class is that of the object to be bound.

The method definition for *bind*, and the exceptions it raises, are as follows:

```
static void bind(naming_context_fp c, string &n, T##_fp &o,
                 naming_context_binding_type bt =
                        naming_context_binding_type_multiple,
                 naming_context_access_list *acl =
                          new naming_context_access_list)
```

```
raises naming_context_not_found,
       naming_context_already_bound,
       naming_context_cannot_proceed,
       naming_context_cannot_proceed_symlink,
       naming_context_failure,
       naming_context_access_denied,
       name_lib_parse_error
              .
              .
```

- *naming_context_fp c*
  The fat pointer for the naming context in which the binding is to be. The binding need not be immediately in this context itself, but may occupy a subcontext at some level below—this should be indicated by the string argument immediately following.

- *string &n*
  A string that specifies the name of the binding. If the binding should occur in a subcontext below the context specified by the naming_context_fp argument, then the string should be a compound name, indicating the true location of the binding.

- *T##_fp &o*
  The object to be bound to the name. The fat pointer must be of the class specified in angle brackets in the naming call. If the binding is successfully achieved, the bound object is no longer immediately available to the principal—the object must be resolved from the Name Service.

- *naming_context_binding_type bt*
  The type of binding to be specified. There are four types—these are discussed below. If not given, this argument defaults to naming_context_binding_type_multiple, which determines that this method hard-links an object, and allows it to be subsequently resolved any number of times by clients.

- *naming_context_access_list *acl*
  The access list that specifies who may resolve or perform other operations on the object. Access lists are discussed below. If not given, this argument defaults to a NULL acl, which grants read-only permission to the world (including the principal who is binding the object).

The *rebind* method is identical to *bind*, except that it does not raise the *naming_context_already_bound* exception—if the object is already bound, *rebind* rebinds it.

An example of the *rebind* method was provided in Chapter 3, *Multiple Domain Programs*:

```
#include "foo_impl.h"
#include "naming/naming.h"
#include "util/basic.h"
#include "util/spring_lib.h"
#include <stream.h>

declare_naming(foo_factory);

main() {
  foo_factory_fp g = foo_factory_fp (*new foo_factory_impl() );

  try {
    naming<foo_factory>::rebind(domain_environment, \
            "village/services/the_factory", g);
  }
  except that {
    default {
      printf("The bind has failed.\n");
        }
  }
  cout << "The bind of object \"the_factory\" has been \
           successfully completed.\n";

  springlib::become_server();

}
```

To enable the appropriate Spring name service processing, any type to be specified in a naming operation must first be declared globally. For this to occur, the type should be provided as the argument to the function *declare_naming*. In the above example, the type of the object to be bound is *foo_factory*, and the call *declare_naming(foo_factory)* thus appears before commencement of the main program.

The static method *rebind* is called with *foo_factory* specified within the angle brackets that follow the class-name *naming* and precede the scope resolution operator. The *naming_context_fp* argument is here given as *domain_environment*, which is the naming context fat pointer provided to every member of a village on start-up—it is the closest thing that Spring has to a *root* context, although no such root truly exists in a distributed environment such as this.

The name to which the object is to be bound is given as the *string* argument: here, it is *"village/services/the_factory"*. The *village/services* contexts are provided as part of the ordered merges that occur when a Spring machine boots, and are shared between participants in the village. Note that if the client had possession of the *naming_context_fp* for the context *village/services* and thus specified such a fat pointer as the line's initial argument, the *string* argument could be shortended to *the_factory*. Examples of this, and an extensive discussion of naming contexts and their fat pointers, will be provided later.

The final argument in the line is the *foo_factory* object to be bound. No binding type or access list is specified. Thus, the defaults are used. The binding type becomes *multiple*, and the acl is *NULL* (see above for descriptions). Once the binding has been achieved, the bound object is no longer immediately available within the domain of the principal: if additional access is required, the object must be resolved from the Name Service.

6.2 *Resolving Objects*

The *naming.h* static method for resolving an object is as follows:

```
static T##_fp resolve(naming_context_fp c, string &n,
                      modes m = modes_default_mode)
            raises  naming_context_not_found,
                    naming_context_cannot_proceed,
                    naming_context_cannot_proceed_symlink,
                    naming_context_cannot_proceed_modes,
                    naming_context_failure,
                    naming_context_access_denied,
                    name_lib_parse_error,
                          .
                          .
                          .
```

- *naming_context_fp c*
  The fat pointer for the naming context in which resolution is to occur. The resolution need not occur immediately in this context itself, but may occur in a subcontext at some level below—this should be indicated by the string argument immediately following.

- *string &n*
  A string that specifies the name of the binding to be resolved. If the binding should occur in a subcontext below the context specified by the naming_context_fp argument, then the string should be a compound name, indicating the true location of the binding.

- *modes m = modes_default_mode*
  The access mode required by the client. If unspecified, the default_mode is provided, which grants the client the highest access permitted, according to the terms of the binding. Modes are discussed in greater detail below.

The following example of *resolve* was given, again, in Chapter 3:

```
for (int i = 0; i < 10; i++) {
    try {
      g = naming<foo_factory>::resolve(domain_environment, \
            "village/services/the_factory");
      break;
    } except that {
      naming_context_not_found {
    (void)printf("Waiting for \"the_factory\" to be published\n");
      }
    }
  }
```

6.3 *Binding and Resolving Contexts*

The procedure whereby a new context is bound is slightly different from that for binding other forms of object. The interface is found in *naming/naming_lib.h*, and appears as follows:

```
static naming_context_fp bind_new_context(naming_context_fp c,
                                          string s,
                                          naming_context_access_list *acl)
        raises  naming_context_not_found,
                naming_context_already_bound,
                naming_context_failure,
                naming_context_access_denied,
                name_lib_parse_error;
```

This static method actually returns the fat pointer for the new context that is being bound. This enables the fat pointer to be reused if necessary. Usage of the *naming_context_fp* and *string* arguments is identical to that already seen in the standard *bind*, *rebind*, and *resolve* calls. Here, the third argument is a

naming_context_access_list. The access list contains entries that specify which users can access and possibly modify the object. The list is defined in *naming/naming_context.idl* as follows:

```
typedef sequence<acl::entry> access_list;
```

Thus, it is a typedef'd *sequence* of *acl_entry* structures, which are themselves defined in *auth/acl.idl* as follows:

```
struct entry_1_0 {
    string name;  // May be a group name
    access::rights rights;
    boolean blacklist;
};
```

The entry is a single grant or denial of access with respect to a specified principal-name. It is a grant of access, unless *blacklist* is set to *TRUE*, in which case it is a denial. The *rights* argument is defined in *auth/rights.idl*. The principal forms of rights, defined within the module *access*, are the following:

- *no_access*
  The principal has no access to the object.
- *read*
  The principal has read-only access to the object.
- *write*
  The principal has write-only access to the object.
- *lookup*
  The principal has.....
- *exec*
  The principal has execute access to the object.
- *read_acl*
  The principal has read-only access to the object's access control list.
- *write_acl*
  The principal has write-only access to the object's access control list.
- *special*
  .....
- *all*
  The principal has all forms of access available to the object and its acl.

An example of creating and binding a new context can be found in the following location:

```
~tjh/spring_progs/chap6/general
```

The file *server.cc* contains the following code, which creates the *naming_context_fp* and specifies its *acl*:

```
main() {

naming_context_access_list *ncal;
ncal = new naming_context_access_list();

struct acl_entry my_entry;
my_entry = acl_entry("tjh", access_all, 0);

ncal->addu(my_entry);

naming_context_fp ncfp;
ncfp = naming_context_fp();
    .
    .
```

A pointer of type *naming_context_access_list* is created and initialized as a new access list sequence. Then, An *acl_entry* structure is created, and initialized with *"tjh"* as the principal, *access_all* (note the standard name-mangling of module-name plus element-name) as the rights, and *0* as the blacklist value. The structure is then added to the *ncal* sequence, using the standard *addu* method provided for sequences. Finally, a *naming_context_fp* is created.

The bind call now proceeds as follows:

```
try {
    ncfp = naming_lib::bind_new_context(domain_environment, \
                        "village/services/my_new_context", \
                        ncal);
    }
    except ex {
        naming_context_not_found {
            printf("naming_context_not_found\n");
        }
        naming_context_already_bound {
                    .
                    .
```

No template is required by this method (or indeed, by any of the methods within *naming_lib.h*). As mentioned previously, the fat pointer is specified as the returned value, rather than as a parameter, of the method.

6.3.1 Resolving and Rebinding a Context

A context can be resolved by means of the same *resolve* method used with all objects. When resolution occurs, the client possesses the appropriate context object and all its state—thus, subcontexts and other objects bound within the resolved context are still contained within it.

Once a context has been bound with *bind_new_context*, it can be resolved and then rebound, under a different name and/or in a different location, using the standard *bind* or *rebind* method from *naming.h*, which was previously examined. Following such rebinding, changes to the state of the context-object—made with reference to any one of its multiple bindings—become reflected in *all* of the bindings. Thus, if context *a* is resolved and rebound as context *b*, and a new object is then bound within *b*, the new object also appears bound within *a*.

Resolving and rebinding a context is demonstrated by a program in the following location:

```
~tjh/spring_progs/chap6/context
```

The file *client.cc* contains the following code, which resolves a context bound by *server.cc* under the name *my_new_context*:

```
naming_context_fp test_ncfp;
try {
test_ncfp = naming<naming_context>::resolve(domain_environment, \
                    "village/services/my_new_context");
} except that {
  default {
    (void)printf("Couldn't resolve my_new_context\n");
  }
}
```

Next, the resolved context object is rebound under the name *further_context*. It is important to note that the *naming_context_binding_type* and *naming_context_access_list* arguments must be specified by the principal when a resolved context object is rebound—the default arguments are not sufficient for binding successfully to occur.

```
naming_context_access_list *my_ncal;
my_ncal = new naming_context_access_list();

struct acl_entry my_entry;
my_entry = acl_entry("tjh", access_all, 0);

my_ncal->addu(my_entry);

try {
  naming<naming_context>::rebind(domain_environment, \
                "village/services/further_context", \
                test_ncfp, \
                naming_context_binding_type_ctx, \
                my_ncal);
}
except ex {
  naming_context_not_found {
    printf("naming_context_not_found\n");
  }
```

Subsequently in this program, a new object is bound within *further_context*, and a *naming_context_binding_iterator* is used to demonstrate that this object automatically becomes bound within *my_new_context* also. See the section *Naming Context Binding Iterators*, below.

6.4 Unbinding

The static method *unbind* is provided by the *naming_lib* class, in *naming_lib.h*. It has the following definition:

```
static void unbind(naming_context_fp c, string s)
                raises naming_context_not_found,
                       naming_context_failure,
                       naming_context_access_denied,
                       name_lib_parse_error;
```

The *naming_context_fp* and *string* arguments are as described for the methods above. The file *client.cc* defines a function that invokes this method:

```
void unbind_from_domain_environment (char *compound_name) {
    try {
        naming_lib::unbind(domain_environment, compound_name);
    }
    except ex {
```

```
naming_context_not_found {
        .
        .
```

Once an object has been unbound from its previously associated name, the object can no longer be accessed under that name.

6.5 *Retrieving Binding-Information*

The interface file *naming_context.idl* provides the definition of a structure that can be used to retrieve information on an object's existing binding:

```
struct binding_info {
    name::component the_name;       // The name in the binding.
    type_id         obj_type;       // The type of the bound object.
    binding_type    bind_type;
    long            create_time;    // Time when object was bound.
    access_list     acl;
    boolean         copy_attribute; // If a context and true, copy
                                    // the contents of the context,
                                    // else copy the context object
    named::named    obj;            // Symbolic link or bound
                                    // object.
};
```

The *binding_info* structure, which is implemented in *naming_context.h* as *naming_context_binding_info*, can be initialized by means of the *naming_lib::get_binding* method, whose definition is as follows:

```
static naming_context_binding_info get_binding(naming_context_fp c,
                                               string s,
                                               bool return_object,
                                               modes m,
                                               bool follow_symlink)
            raises  naming_context_not_found,
                    naming_context_failure,
                    naming_context_access_denied;
```

- *naming_context_fp c*
  The fat pointer of the context in which the binding resides.
- *string s*
  A string that is the name of the binding on which information is required.

- *bool return_object*
  If *TRUE*, return the object itself, as if by a *resolve*. If *FALSE*, do not return the object. Returning the object does not affect the binding, which continues to exist. Changes made to the object via any binding will immediately be reflected in the state of the object as perceived via all other bindings and via the object returned by means of the *binding_info* structure.

- *modes m*
  .....

- *bool follow_symlink*
  If the specified binding is a symbolic binding and *follow_symlink* is *TRUE*, follow the symbolic binding and retrieve information on the object to which the symbolic binding leads. If *follow_symlink* is *FALSE*, do not follow the symbolic binding—instead, return information on the object that is the symbolic binding itself.

An example of binding information retrieval is found in the following location:

```
~tjh/spring_progs/chap6/general
```

The file *client.cc* contains the following code, which demonstrates how to create a *naming_context_binding_info* structure and initialize it with the *get_binding* static method:

```
struct naming_context_binding_info my_info;
my_info = naming_lib::get_binding(domain_environment, \
    "village/services/my_new_context", 1, modes_default_mode, 0);
```

This specifies that a symbolic link should not be followed (indeed, the object referenced is a context that has not been symbolically linked) and that the object itself should be returned in the appropriate field of the structure.

Each field of the *naming_context_binding_info* structure can now be discussed in relation to the code used in *client.cc* to initialize and examine it.

- *name::component the_name*
  A *name_component_fp*, which contains the name of the bound object. Name components are defined in *name.idl*, and are more for the use of internal name service processes, rather than of client code. Name components are discussed in detail below, in *Using Names and Strings*.

  The method *get_id* is defined on name components, to allow return of the string that is the name represented by the name component. In this case, the name is that of the bound object:

```
name_component_fp ncomp;
ncomp = my_info.the_name;
string str;
str = ncomp->get_id();
cout << "\n" << "Client now knows ";
cout << "the name of the context is " << str.s2c();
cout << "...\n";
```

- *type_id obj_type*
  The type identifier of the object. The *type_id* is a C++ object defined in *type_service/type_id.h*. The class defines a *human_readable* method, which can be used for examination of the identifier:

```
type_id tid;
tid = my_info.obj_type;
str = tid.human_readable();
cout << "\n" << "... and its type id is " << str.s2c();
cout << ".\n";
```

- *binding_type bind_type*
  The binding type of the object. This is defined in *naming_context.idl* within the *naming_context* module, as follows:

```
enum binding_type {
    ctx,            // Object being bound is a context.
    symbolic_link   // Object being bound is a symbolic link.
    unique,         // When a name is resolved to this binding,
                    // the original copy of the object is
                    // returned, and the binding removed.
    multiple        // When a name is resolved to this binding,
                    // a "copy" of the object is returned.
};
```

  After name mangling, the tags of this Contract enum are implemented as *naming_context_binding_type_ctx*, etc.

  As the comments in the file indicate, the enum permits indentification of the type of binding that exists. The following additional notes are provided:
  - If a binding is *unique*, resolution of the object removes the binding—thus, typically, objects are bound with the *multiple* binding type.
  - A *symbolic_link* is a form of object that, when followed, may lead to a context or non-context object, the binding of which may be either *unique* or *multiple*. A *symbolic_link* is itself invariably bound with the *multiple* binding type.

- A context (*ctx*) object is also invariably bound with the *multiple* binding type.

- *long create_time*
  The time at which the object was bound.....

```
int ct;
ct = my_info.create_time;
cout << "\nThe context's create time was " << ct << ".\n";
```

- *access_list acl*
  The object's access list.....(permissions?)

```
naming_context_access_list *my_list;
my_list = my_info.acl;
struct acl_entry an_entry;
an_entry = my_list->elem(0);
str = an_entry.name;
cout << "\nThe name entry in the object's ";
cout << "access list is " << str.s2c() << ".\n\n";
```

- *boolean copy_attribute*
  If a context and true, copy the contents of the context, else copy the context object.

- *named::named obj*
  A *named* fat pointer that relates to the bound object. The *named* interface is used internally by the name service in the coordination between client-specified objects and the name objects that must represent them. Typically, the *named* interface (like the *name* interface mentioned above) should not be used by client programmers.

  The *named* fat pointer that is returned in this field can be manually translated to the client-specified object that it represents. This is demonstrated by the following code, found in *client.cc*:

```
named_fp nfp;
nfp = my_info.obj;
named_of_fp<naming_context> y;
y = named_of<naming_context>::narrow_it(nfp);
naming_context_fp ncfp;
ncfp = y->unwrap();
```

  Though the precedent be a bad one, the exact details of these interfaces, and meaning of thie above code, will not be explored in this document—save to note that *naming_context* must be declared in a *declare_naming* call in order to be used in the given template.

6.5.1 Naming Context Binding Iterators

A *naming_context_binding_iterator* is the C++ implementation of the *binding_iterator* class, defined in the module *naming_context* within the file *naming_context.idl:*

```
typedef sequence<binding_info> binding_list;
interface binding_iterator {
    void next(copy long how_many,
              produce binding_list bindings,
              produce boolean more);
};
```

The *how_many* parameter describes the number of iterations that are required over the bindings in a given context. The *bindings* parameter is a *binding_list*, which is a *typedef*'d *sequence* of *binding_info* structures (see *Retrieving Binding Information*, above). The *more* parameter is a *boolean*, which will indicate whether there were additional bindings in the context that were not iterated over (that is, if it is 1, the number of iterations specified by the *how_many* parameter was less than the number of bindings in the context).

Since the *bindings* and *more* parameters are specified as being in the *produce* parameter passing mode (see *Parameter Passing Modes*, in Chapter 5, *Essential Spring Features)*, their C++ implementations will be as *reference* (or *out*) parameters.

Use of an iterator is demonstrated in the following location:

```
~tjh/spring_progs/chap6/general
```

Note – Erratum. The following description relates to a program no longer included. However, the substance of the description is not inconsistent with the content of the *general* program, cited correctly in the pathname above. I'll make the appropriate modifications to the passage below in the immediate future.

Here, iterators are used over a number of contexts. In particular, they demonstrate how binding an object within a context that has been successively resolved and rebound (and thus has two current and distinct bindings) causes the object itself to be reflected as a binding within both of the contexts. The iterator is used as follows:

```
naming_context_binding_iterator_fp itfp;
```

```
itfp = naming_lib::get_all_bindings(domain_environment, \
                                    "village/services/my_new_context", 0);
naming_context_binding_list *nlist;

  itfp->next(-1, nlist, 1);
  for (int i = nlist->lwb(); i <= nlist->upb(); i++) {
    string a_string = (*nlist)[i].the_name->get_id();

    cout << "Object name ";
    cout << i;
    cout << " from iterator is \"" << a_string.s2c() << "\".\n";
    cout << "\n";
  }
```

The fat pointer variable is defined, and is then initialized by invocation of the the static method *naming_lib::get_all_bindings*. The first two arguments to this method are the familiar *naming_context_fp* and *string*, which together identify the context object whose bindings are to be inspected. The final argument is a *bool*, which indicates whether or not to return the bound object (as if by means of a *resolve*).

The *next* method of the created iterator is then invoked. Normally, the first of its three arguments should be a positive integer, indicating the required number of iterations. In this case, however, *-1* is used: this is apparently a hack, used to return *all* bindings in the context. The second argument is a pointer variable of type *naming_context_binding_list*. The third argument is the boolean, which will indicate whether additional bindings exist following execution (since *-1* has been specified, the boolean will be 0, indicating that all bindings have been returned).

Since the *naming_context_binding_list* is actually a *typedef*'d *sequence*, the standard *sequence* methods *lwb*, *upb*, and *operator[]* can be used to process the contents of the binding list, following execution of *next*.

6.6 Freezing and Melting

Interfaces are provided for *freezing* and *melting*. These features are normally used only by the internals of the name service. However, since they allow an object to be passed between domains in a non-standard form, they might occasionally be useful to developers.

A demonstration of how to use these interfaces can be found in the following location:

`~tjh/spring_progs/chap6/freeze`

This directory contains three main programs. The first, named *server*, simply creates a *foo_factory* and binds it to a name. The second, named *freezer*, resolves the factory and freezes it—the results of the freeze operation are a *freeze token* and an *omid* (object manager id); *freezer* then initializes a data structure-like object with these values, and binds the object to a name. The third program, *client*, resolves the object bound by *freezer*, *melts* the *freeze token* that it retrieves from the data structure, and invokes methods on the resulting *foo* object.

Thus, an object has been passed between programs in the form of a token and identifier pair. In fact, since the *client* program must use the name service to resolve the data structure, and since the *client* must also contain all appropriate *includes* and must be properly linked in order to use the melted *foo* object, nothing is gained by the exercise, save a demonstration of a facility typically used by the name service in binding and rebinding objects. (The name service accepts a token and identifier from each object manager that wishes to bind an object; at resolve time, it uses the token and identifier to tell the object manager to provide a corresponding object to the client that is attempting the *resolve* operation.)

In this example, the file *freeze_server.cc* binds the object *the_freezer*, which is defined in *foo_impl.h* to return a data structure object, defined as the class *data_impl*. The *data_impl* object is designed to contain the token and identifier of a frozen object. When *client.cc* has resolved *the_freezer*, it calls the *get_data* method, which causes the *freezer* program to *(a)* resolve a *foo_factory*, *(b)* create a *foo*, *(c)* freeze the *foo*, and *(d)* appropriately initialize and return a data structure object to the *client*.

The interfaces for freezing and melting are provided in *naming/freeze_lib.h*.The *freeze* interface, used by the file *freezer_impl.cc* to freeze the resolved *foo_factory* object, is as follows:

```
static freeze_token *freeze(named_fp &object, string &om_id)
    raises  freeze_cannot_find_freeze_service,
            freeze_cannot_freeze,
            contract_fault;
```

The parameters to this method are respectively a *named_fp* and a *string* that is the object manager's id. The id is an *out* parameter: thus, an empty string can be passed, and will be initialized by the method. By contract, the *named_fp* must be specified. The *named* interface, defined in *naming/named.idl*, is another of the interfaces used principally by the internals of the name service for

handling the relationship between client-defined objects (which the client program is interested in) and name object (which are the only objects that the name service itself actually handles). Once a *foo_factory* has been resolved in the expected way, and a *foo* object called *alpha* created, *freezer_impl.cc* uses the following code to create the appropriate *named_fp*:

```
named_of_fp<foo> named_of_alpha;
named_of_alpha = named_of<foo>::wrap(alpha);

named_fp named_alpha;
named_alpha = named_of_alpha->dup();
```

As in the previous example regarding wrapping and unwrapping, this code will not be discussed in detail, since it is not typically for the use of the developer.

Once an appropriate data structure object and an empty string have been created, the call to the *freeze* method occurs:

```
data_fp data_fp (*new data_impl() );
    .
string srvr_omid;
    .
try {
    dfp->set_token(freeze_lib::freeze(named_alpha, srvr_omid));
}
except cept {
    freeze_cannot_freeze {
            .
            .
```

This uses the *freeze* method to set the *token* field of the data structure object that will be returned to the *client*. Subsequently, the *srvr_omid* can also be used to initialize this data structure:

```
dfp->set_om_id(srvr_omid);
```

The file *freezer_impl.cc* shows how the bound *foo_factory* object is resolved and frozen

```
a_fac = naming<foo_foo_factory>::resolve(domain_environment, \
    "village/services/my_new_context/the_factory");
            .
            .
```

The *client.cc* file contains the following code, which invokes the *melt* method on the retrieved token and id, and then unwraps the resultant *named_fp*:

```
named_fp nfp
```

```
try {
    nfp = freeze_lib::melt(retrieved_om_id, retrieved_ft, \
                modes_default_mode, 0);
}
                        .
                        .
named_of_fp<foo> y;
y = named_of<foo>::narrow_it(nfp);
foo_fp new_gamma;
new_gamma = y->unwrap();
```

Finally, *client.cc* invokes the methods of the unwrapped object in the expected way.

6.7 *Access Control Lists and Authorization*

To be added.

6.8 *Symbolic Linking*

The Spring name service permits symbolic links to be created to objects (including context-objects), provided that each object to be symbolically linked resides within or below the context from which the symbolic link itself proceeds. A demonstration of symbolic linking is found in the following location:

`~tjh/spring_progs/chap6/symlink`

In this two-domain program, the file *server.cc* creates and bind two contexts, named *upper* and *lower*. It then binds an object named *foo_factory* within *lower*. Finally, it creates a symbolic link from *upper* to *foo_factory*. The code for doing this is as follows:

```
foo_factory_fp h = foo_factory_fp (*new foo_factory_impl() );

try {
    naming<foo_factory>::rebind(ncfp, \
                                "lower_context/factory_object", \
                                h);
                    .
                    .
naming_context_access_list *fourth_ncal;
fourth_ncal = new naming_context_access_list();
```

```
fourth_ncal->addu(my_entry);

try {
    naming<name>::rebind(ncfp, \
        "factory_symlink", \
        name_lib::string_to_name("lower_context/factory_object"), \
        naming_context_binding_type_symbolic_link, \
        fourth_ncal);
```

The *foo_factory* is bound in the standard way, using the *rebind* static method, provided by the *name.h* interface. The symbolic link is also bound with this method, an invocation of the *string_to_name* method (from *name_lib.h*) being specified as the object argument—the method provides a *name_fp*, and it will indeed be to this name that the symbolic link is made. The binding type is naturally specified as *naming_context_binding_type_symbolic_link*. (Note that while this symbolic link, proceeding from *upper* to *lower*, is perfectly feasible, no such link could have been made from *lower* to *upper*. The object that is the ultimate target of the symbolic binding must be either in the same context as that from which the symbolic link proceeds, or must be somewhere below it.)

As the above code demonstrates, while *hard* bindings are typically the association of a name with an object, a *symbolic* binding is the association of one name (that of the symbolic binding itself) with another (that of a hard-bound object). When the name of a symbolic binding is referenced, it may therefore be necessary to determine whether the name is to be *followed* (in which case an operation can be performed on the object bound to the name to which the symbolic link leads) or *returned* (in which case the object and its hard binding are unaffected). Thus, an operation equivalent to the UNIX *rm* command, with a symbolic link as its argument, should be expected to remove the symbolic link, not the object to which it and possibly many other symbolic links lead—in this case, then, the symbolic link should be *returned*, not *followed*.

The file *client.cc* demonstrates how the distinction between returning and following a symbolic link can be made by using the *get_binding* static method, provided by *naming_lib.h*. The final argument to this method, *follow_symlink*, is a boolean. If the specified binding is a symbolic binding and *follow_symlink* is *TRUE*, the symbolic binding is followed, and information is retrieved on the object to which the symbolic binding leads. If *follow_symlink* is *FALSE*, the symbolic binding is not followed—instead, information is returned on the object that is the symbolic binding itself.

Initially, *client.cc* retrieves the object itself:

```
struct naming_context_binding_info my_info;

my_info = naming_lib::get_binding(domain_environment, \
            "village/services/upper_context/factory_symlink", \
            0, \
            modes_default_mode, \
            1);
```

Then, the object's name, type_id, and binding type are retrieved from the *my_info* structure. The resultant output is as follows:

```
Client now knows the name of the object is factory_object...

..and its type id is 0x0E...66163746F7279 ("::foo_factory").

Its bind type is 3.
```

Subsequently, *client.cc* retrieves the symlink, rather than the object itself, by specifying 0 as the final argument to *get_binding*:

```
    my_info = naming_lib::get_binding(domain_environment, \
        "village/services/upper_context/factory_symlink", \
        0, \
        modes_default_mode, \
        0);
```

This is duly reflected in the program's output:

```
Client now knows the name of the object is factory_symlink...

..and its type id is 0x436008A

Its bind type is 1.
```

6.9 *Using Names and Strings*

As described above, the Name Service makes frequent internal use of *name* and other related fat pointers. Although these typically should not be used by developers, they are nevertheless discussed in this section, since they may need to be dealt with from time to time. The principal interfaces involved are *name*, which defines the standard Spring object for representing a name, and *name_components*, which defines components that can be assembled into a name. These interfaces are both described, as are various methods contained in *name_lib.h*, which allow *name* (or *name component*) fat pointers to be converted easily to *strings*, and vice versa.

A demonstration of the use of *name* fat pointers is found in the following location:

```
~tjh/spring_progs/chap6/name
```

6.9.1 Introduction to Strings, Names, and Name Components

In Spring, *strings*, *names*, and *name components* can be used interchangeably. The C++ interface for strings is provided by *util/string.h*, that for names and name components (as for all aspects of naming) by *naming/naming.h*

As repeatedly demonstrated, the typical and advisable way of specifying a name in Spring is by means of a string. Thus, the following *rebind*, found in *client.cc*, provides a compound name as a string, indicating the name to which the object *gamma* is to be bound:

```
naming<foo>::rebind(domain_environment, \
    "village/services/a_new_context/foo_object", gamma);
```

As an alternative, a *name* fat pointer can be specified. The easiest way to create such a fat pointer is by means of the *string_to_name* static method, provided by *name_lib.h*—this include file provides a variety of static methods for conversion between strings, names, and name components. The method for converting a *string* to a *name* is as follows:

```
name_componet_fp ncompx;
ncompx = name_lib::string_to_name(\
    "village/services/a_new_context/foo_object");
```

Thus, the *rebind* call would now appear as shown below:

```
naming<foo>::rebind(domain_environment, ncompx, gamma);
```

Or, conversely, if the client is in possession of a *name* that it wishes to specify as a *string*, the *name_lib.h* static method *name_to_string* could be used. This is demonstrated in *client.cc* for a *name* fat pointer called *my_name*:

```
naming<foo>::rebind(domain_environment, \
    name_lib::name_to_string(my_name), gamma);
```

In the Spring Name Service, most of the standard methods used to *bind*, *rebind*, and *resolve* objects are defined in two ways, one accepting a *string*, the other a *name*. Thus, *strings* and *names* are highly interchangeable.

Note that a *name* object can be accessed only prior to being used in a *bind* or *rebind* call. Once it has been so used, a segmentation fault results from further attempts to access it.

A *name component* is an object that represents a component of a name—that is, it represents a single, simple name that may or may not be part of a compound name. Thus, in the compound name *village/services, village* is one component, and *services* another. *Name components* can be created individually and appended, inserted, etc. to form *names*. Within *client.cc*, the following code is used to demonstrate how a *name* might be built in this way:

```
name_component_fp ncomp1;
ncomp1 = name_lib::string_to_component("village");

name_component_fp ncomp1a;
ncomp1a = name_lib::string_to_component("services");

name_component_fp ncomp2;
ncomp2 = name_lib::string_to_component("a_new_context");

name_component_fp ncomp3;
ncomp3 - name_lib::string_to_component("foo_object);

name_fp my_name;
my_name = name::from_component(ncomp1);
my_name->append_comp(ncomp1a);
my_name->append_comp(ncomp3);
my_name->insert_comp(2, ncomp2);
```

Here, each component is created by means of the *string_to_component* library method, from *name_lib.h*. The name *my_name* is then initialized with the static method *from_component*, defined by *name.idl*. Subsequently, other components are appended and inserted as appropriate.

When using name components, developers should always ensure that each component indeed represents only a single, simple element of a compound name. Though it is possible to specify an entire compound name as a component, this is not legitimate practice. Additionally, compound names specified in this way become incapable of modification—no other components can be added to them in a way that results in a successful search.

Note that, like *names, name_components* are inaccessible once they have been used (within a given *name*) to perform a binding. Note also that a given *name_component* should only be used with one *name*. Segmentation faults occur when *names* attempt to share components.

6.9.2 *Methods on Name Components*

Within *name.idl*, the *component* interface is defined as follows:

```
interface component {

    class component make(
        copy string id,
        copy long version,
        copy comp_kind k
    );

    string get_id();
    void set_id(copy string id);
    long get_version();
    void set_version(copy long version);
    comp_kind get_kind();
    void set_kind(copy comp_kind kind);

    boolean equal(borrow component comp);
    boolean match(borrow component_expression comp);

    long hash(copy long max);
};
```

The C++ implementations of the Contract methods are described as follows:

- *name_component_fp make(string arg_id, int arg_version, name comp_kind k arg_k)*
  The constructor returns a name component with the specified identifier, version, and kind. The identifier is the *string* that specifies the simple name represented by the component. The version represents the Spring version number (if any) of the object—this should only be used if a version number actually exists and is routinely used as part of the simple name of the object. The *kind* must currently be *generic*—no other kinds have yet been defined.

- *string get_id()*
  Returns the *string* that is the simple name represented by the component. Thus:

```
string id_string;
id_string = ncomp4->get_id();
printf("\nThe id is %s.n", id_string.s2c());
```

- *void set_id(string arg_id)*
  Sets the identifier of the component to the specified string. Note that if a component is currently used by one or more *name* fat pointers, and the identifier of the component is modified with *set_id*, this change is reflected in the contents of all the *name* fat pointers.

- *int get_version()*
  void set_version(int arg_version)
  Return and set the version number of the object.

- *name_comp_kind get_kind()*
  void set_kind(name_comp_kind arg_kind)
  Return and set the *kind* of the component. Currently, only *name_comp_kind_generic* is permissible. Thus:

```
ncomp4->set_kind(name_comp_kind_generic);
```

- *bool equal(name_component_fp &arg_comp)*
  bool match(name_component_expression_fp &arg_comp)
  Return *TRUE* if the argument is equal to this component, *FALSE* otherwise. A *name_component_expression* is a *typedef*'d *name_component*.

- *int hash(int arg_max)*
  Returns the result of executing a hash function on the component. The returned value is in the range *0 <= value <= max*.

6.9.3 Methods on Names

The file *name.idl* defines the *name* interface as follows:

```
interface name {
    class name from_component(consume component c);
    class name null_name();

    long length();
    component get_comp(copy long n)
        raises (bad_index);
    component last_comp()
        raises (empty);
    void set_comp(copy long n, consume component comp)
        raises (bad_index);
    void append_comp(consume component comp);
    void insert_comp(copy long n, consume component comp)
        raises (bad_index);
    component delete_comp(comp long n)
```

```
        raises (bad_index);
    boolean equal_comp(copy long n, borrow component comp)
        raises (bad_index);
    void concat(consume name concat_name);
    name base(copy long n)
        raises (bad_index);
    name rest(copy long n)
        raises (bad_index);
    boolean equal(borrow name n);
    boolean subname(borrow name n);
    long hash(copy long max);
};
```

The C++ implementations of these methods are described below.

- *static name_fp from_component(name_component_fp arg_c)*
  This constructor returns a *name* fat pointer initialized exactly as is the *name* fat pointer specified as the argument.

- *static name_fp null_name()*
  This constructor returns an empty *name* fat pointer.

- *int length()*
  Returns the number of components in the name.

- *name_component_fp get_comp(int arg_n)*
  Returns a fat pointer for the component that is at the given index position. Thus:

```
string str;
name_component_fp some_component;
some_component = my_new_name->get_component(2);
str = some_component->get_id();
cout << "Third component is ";
cout << str.s2c() << ".\n";
```

Note – The object returned by *get_component* is a *deep* copy of the original object. Thus, the copy can be modified independently of the original, nor does the modification affect any existing *name* object that uses the original as one of its components.

- *name_component_fp last_comp()*
  Returns the last component within a name.

- *void set_comp(int arg_n, name_component_fp arg_comp)*
  Sets the component at index position *arg_n* to be the fat pointer specified by *arg_comp*. The previous occupant of the index position is deleted from the *name*'s component list.

```
name_component_fp latest_component;
latest_component = /
        name_lib::string_to_component("latest_component");
my_new_name->set_comp(2, latest_component);
```

- *void append_comp(name_component_fp arg_comp)*
  Appends the given component to the end of the component list.

- *void insert_comp(int arg_n, name_component_fp arg_comp)*
  Inserts the given component at the specified index position. Where necessary, existing components are moved back by one index position, to make room for the inserted component.

- *name_component_fp delete_comp(int arg_n)*
  Deletes the component at the specified index position. Where necessary, remaining components are moved forward by one index position, to close the space left by the deleted component.

- *bool equal_comp(int arg_n name_component_fp &arg_comp)*
  Tests for equality between the component at the specified index position and the component specified as the *arg_comp* argument. All fields of the components must be identical for *TRUE* to be returned.

- *void concat(name_fp arg_concat_name)*
  Concatenates the name components of the specified *name_fp* onto this *name_fp*'s component list. Following concatenation, the fat pointer that was specified as the *arg_concat_name* is deleted.

- *name_fp base(int arg_n)*
  Returns a new *name_fp* that contains deep copies of the name components of this *name_fp*, from index position 0 to the index position specified by *arg_n*.

- *name_fp rest(int arg_n)*
  Returns a new *name_fp* that contains deep copies of the name components of this *name_fp*, from index position *arg_n* to the end.

- *bool equal(name_fp &arg_n)*
  Returns *TRUE* if this *name_fp* is equal to the one specified as *arg_n*, *FALSE* otherwise.

- *bool subname(name_fp &arg_n)*
  Returns *TRUE* if the component list of the *name_fp* specified as *arg_n* represents a subname of the list of this *name_fp, FALSE* otherwise.
- *int hash(int arg_max)*
  Returns the result of executing a hash function on this *name_fp*. The result is a value in the range *0 <= value <= max*.

6.9.4 Methods in name_lib.h

To be added.

6.10 Duplication

To be added. A weak example is in *~tjh/spring_progs/chap6/dup*.

6.11 Ordered Merges

To be added. A reasonable example is in

`~tjh/spring_progs/chap6/om`

Index

Symbols

A

B

E

F

I

L

M

N

T

V

W