UKC Orbix Survival Guide

I.C.A.Buckner & I.A.Penny
Computing Laboratory, University of Kent at Canterbury

May 6, 1997

1 Introduction

So you need to write an Orbix application but don't know where to start? This document will try to break you in as gently as possible by explaining the concepts you need to understand and then show you how Orbix implements them. If you read it in conjunction with the programmer's guide [Iona, 1995a] then you should get a good idea of what is going on. The concepts and much of the implementation details are generic but there are parts which are specific to UKC, and we will highlight these when we come to them.

2 Concepts

In order to understand the fundamentals of Orbix there are two concepts you must be familiar with: distributed systems and object orientation. There are numerous books on these subjects, so we will limit ourselves to a brief introduction.

2.1 Distributed systems

In a traditional, non-distributed, system, when you wish to solve a problem you write a program which starts, calculates the solution and then stops. It is highly likely that you divide your program into a number of sub-tasks, each implemented by a separate function, which, when executed in turn, perform your required task.

In a distributed system we build a solution to a particular problem by building a number of programs which when they communicate perform the required task. The programs can be distributed across a network of machines to take advantage of the resources of each one; this distribution should be transparent to the user. In theory it should not matter if the machine is in the next room or on the next continent, although there are obvious performance advantages in having the machines contained within a local network.

2.2 Object orientation

We are surrounded by objects in everyday life, some simple, others complex. Often complex objects are created from a collection of other, simpler, objects, thus hiding the complexity of the inner workings from us. For example a watch consists of a number of cogs, hands, and springs; we do not interact with these components we interact with the watch as an object in its own right.

In object oriented computing "the underlying concept is that one should model software systems as collections of cooperating objects" [Booch, 1994]. Server objects provide operations which may be invoked by client objects to elicit some change in state; Booch identifies five distinct operations :

- A **Modifier** alters the state of an object.
- A **Selector** accesses the state of an object, but does not alter the state.
- An **Iterator** permits parts of an object to be accessed in a well-defined order.
- A **Constructor** creates an object and initialises its state.
- A **Destructor** frees the object's state and destroys it.

The implementation of an operation within a server object is transparent to client objects. This division between implementation (definition) and interface (declaration) provides a clear boundary to the object which is associated via the interface.

3 Writing Orbix programs

Orbix is an implementation of the Common Object Request Broker Architecture (CORBA) [OMG, 1991]. It combines distributed systems with object orientation to provide a distributed programming environment in which a system can be constructed from a number of communicating objects. The communication mechanism is provided by the environment; the role of the programmer is to design and implement one or more server objects, and a client which invokes the server.

This section describes, in detail, the steps required to write a client and server using Orbix; it is supplemented by a checklist which can be used in your future projects. We shall examine the process by working through an example system, from interface description to compilation of the code. The code used in this example is available online in the directory `/proj/orbix/doc/survival/src`.

3.1 Problem description

"A retail system comprises a component which maintains a record of stock item serial numbers, prices, and descriptions. The serial number and price are of type long. The description is a character string. Operations are required to register an item, delete an item, find its price, change its price, find its description, and change its description."

3.2 Interface description

Before we actually write the code for the server object we must specify the interface which will be used by the client when it wishes to invoke server operations. To do this you must write an interface description language (IDL) file; this file is processed by Orbix's IDL compiler. The retail system, described previously, will use the following IDL file :

```
// In file retail.idl
interface stock {
        // Attributes
        readonly attribute long item_count;

        // Functions
        void newitem(in long serial, in long cost, in string name);
        void delitem(in long serial);

        long getprice(in long serial);
        void setprice(in long serial, in long cost);

        string getname(in long serial);
        void setname(in long serial, in string name);
};
```

We see that there are six functions, four of which return no result and the others return a long integer and a character string respectively. It is also possible to pass these results back as *out* parameters, but we prefer to return them as the result of the function.

3.3 Compiling idl

To compile the IDL file `retail.idl` you must type the following :

```
kestrel% idl -B -S retail.idl
```

The -B flag states that you wish to use the BOAImpl approach, see section 1.4.1 of the programmer's guide for details. The -S flag asks the idl compiler to drop skeleton code for the declaration and definition