Object Oriented Programming Techniques Applied to Device Access and Control

A.Götz, W.D.Klotz, J.Meyer
ESRF, BP 220
F-38043 Grenoble Cedex
FRANCE

1 Introduction

Device access and device control is one of the most important tasks of any control system. This is because control implies obtaining information about the physical world by reading sensors and modifying the behaviour of the physical world by sending commands to actuators. At the **European Synchrotron Radiation Facility (ESRF**, ref. [1]) effort has gone into designing and implementing a model for device access and control using as much as possible the latest ideas and methods of Software Engineering. One of the main contributions in recent years to Software Engineering has been in the field of **Object Oriented Programming(OOP)**. Although the philosophy is not new the refinement and application of this methodology on a wide scale is. At the ESRF a model for device access and control has been developed which is based on **OOP** methods. This model, called the **device server model**, is the topic of this paper. The device server model is written entirely in C and is therefore portable. It depends on no other software and can be ported to any machine where there is a C compiler. Because the model is based on **OOP** it presents a *user-oriented* view of the world as opposed to a *software-* or *hardware-oriented* view of the world.

This paper will describe the device server model. It will describe the problem of device access and the advantages of using **OOP** techniques to solve it. It will present the model. The methodology used to implement **OOP** in the device server model called **Objects In C (OIC)** will be described. An example of a typical device server at the ESRF will be presented. The experience gained from the device server model will be discussed. The paper will conclude with a discussion on how the device server model could be standardised to treat a wider range of problems.

2 The Device Access Problem

The problem which the device server model is designed to solve is a problem which every control system is faced with. The problem could be described as - *how to provide access and control for all the physical devices which represent the machine ?*

Unfortunately there is no widely accepted industrial standard for interfacing devices to computers. Although some attempts have been made at defining an industrial standard none of them have succeeded (see [2]). This means that there are about as many ways to interface a device to a computer as there are device suppliers.

The device access problem would be simple if a single standard were adopted for interfacing. In reality however this turns out to be too expensive because it involves extra development costs for the suppliers.

3 The Device Server Model

At the ESRF a unified model (called the **device server model**) has been developed to solve the problem of device access and control. It is unified for two reasons —

- it presents a single interface for upper level applications to all kinds of devices, and
- it defines the framework within which to implement device access and control for all devices.

The model can be divided into a number of basic elements - the *device*, the *server*, the *Objects In C* methodology, the *root class*, the *device class*, *resource database*, *commands*, *local access*, *network access*, and the *application programmers interface*.

3.1 The Model

The basic idea of the device server model is to treat each **device** as an **object** which is created and stored in a process called a **server**.

Each device is a separate entity which has its own *data* and *behaviour*. Each device has a *unique name* which identifies it in network name space. Devices are configured via **resources** which are stored in a **database**. Devices are organised according to classes, each device belonging to a class. Classes are implemented in C using a technique called **Objects In C**. All classes are derived from one **root class**. The class contains a generic description of the device i.e. what actions can be performed on the device and how it responds to them. The actions are made available

via commands. Commands can be executed *locally* or *remotely* (i.e. across a network). Network access to a device and its commands is provided by an application programmers interface using a remote procedure call.

3.2 The Device

The device is at the heart of the device server model. It represents a level of abstraction which previously did not exist. A device can be a physical piece of hardware (e.g. an interlock bit), an *ensemble* of hardware (e.g. a screen attached to a stepper motor), a logical device (e.g. a taper), or a combination of all these (e.g. a storage ring). Each device has a unique name. At the ESRF a three field name space (consisting of DOMAIN/FAMILY/MEMBER) has been adopted.

The decision of what level of abstraction a device represents depends on the requirements of the clients. At the ESRF these are the machine physicists. Devices should model the clients view of the problem as closely as possible. Hardware dependencies should be hidden behind the device abstraction. For example if a corrector consists of three independant powersupplies the client (assuming she is a machine physicist) should only see a single device — the corrector, and not three independant devices.

All devices are treated as having state. Each device has a list of commands which it understands. Before any command can be executed the state machine gets checked to see if the command can be executed in the present state. The commands and the state machine are implemented in the device's class.

3.3 The Server

Another integral part of the device server model is the server concept. The server is a process whose main task is to offer one or many *service(s)* for *clients* who want to take advantage thereof. The server spends most of its time in a *wait-loop* waiting for clients to demand its service(s). This division of labour is known as the client-server concept. It exists in different flavours and is very common in modern operating systems.

The adoption of this concept in the device server model has certain implications. The server in the device server model is there to serve one or many device(s) i.e. there is one server per device but there can be many devices per server. The exact configuration depends on the hardware organisation, CPU charge, and the available memory. The fact that there can be many devices per server means that a single device should not monopolise the server for more than a pre-defined amount of time. Otherwise the server is blocked and new or existing clients will not be able to connect to the same or other devices served by that server. The server waits for clients by listening at a certain network address. The mechanism for doing this is implemented by a remote procedure call. At the ESRF the network addresses are determined dynamically (at server startup time) and then stored in a *database*. The first time a client connects to a server it goes to the database to retrieve the server's address, after which it communicates directly with the server.

3.4 Objects In C

The use of objects and classes in the device server model necessitates appropiate OOP tools. The natural choice would have been to use one of the many OOP languages which are available on the market today. If possible one for which a standard exists or will exist (e.g. C++). The choice of a language is not independant of the development evironment however. The language chosen has to be fully compatible with the development environment. At the ESRF the device access and control development environment consists of OS9, HP-UX, SunOS and the SUN NFS/RPC. Unfortunately there is no commercially available OOP language compatible with this environment. The only language which supports the above environment is C. In order to use OOP techniques it was therefore necessary to develop a methodology in C, called Objects In C (from here on OIC). The methodology developed is implemented entirely in C and is closely modelled on the widget programming model (ref. [3]).

OIC implements each class as a structure in C. Class hierarchies are supported by subdividing a class structure into *partial* structures. Each partial structure representing a super- or a sub-class. Each class requires a minimum of three files :

- a private include file describing the class and object structures,
- a public include file defining the class and object types as pointers to structures and the class as an external pointer to the class structure,
- a source code file which contains the code implementing the class.

The private include file is used to define constants and/or variables which should not be visible to the outside world (the inverse being true for the public include file). All functions implementing the class are defined to have static scope in C. This means that they cannot be accessed directly by any other classes or applications – they are only accessible via the method-finder or as commands. This enforces code-hiding and reinforces the concept of encapsulation – a way of reducing coupling between software modules and making them immune to changes in the class implementation.

OIC implements an explicit method-finder. The method-finder is used to search for methods in a class or hierarchy of classes. The method-finder enables methods to be inherited. Two special versions of the method-finder exist for creating and for deleting objects.

Objects are also implemented as structures. Each object has a pointer to its class structure. This means that class

related information (data and code) are stored only once per process for all objects of the same class.

3.5 The Root Class

All device classes are derived from the same root class, the **DevServerClass**.

The DevServerClass contains all common device server code. This includes all code related to the application programmer interface, the database connection, security, administration and so on. All device classes inherit this code automatically, this means improvements and changes are inherited too. The decision to have a single root class from which all other clases are derived has been fundamental to the success of the device server model. Other **OOP** based systems have used the same principle (e.g. the **NIH** set of classes, see [4]).

3.6 The Device Class

Organising devices into classes is an attempt to generalise on common features between devices and to hide device dependant details. The device class contains a complete description and implementation of the behaviour of all members of that class. Hardware specific details are implemented by the class in such a way that they are transparent to device server clients. Typically the first level of device classes (i.e. classes which deal directly with the first level of hardware) will use the utilities offered by the operating system (e.g. device drivers) to implement the hardware specific details.

New device classes can be constructed out of existing device classes. This way a new hierarchy of classes can be built up in a short time. Device classes can use existing devices as **sub–objects**. This means they appear as terminators in the class structure and not within the class hierarchy. This approach of reusing existing classes is classical for **OOP** and is one of its main advantages. It encourages code to be written (and maintained) only once.

3.7 The Resource Database

Implementing device access in classes forces the programmer to implement a generic solution. To achieve complete device independance it is necessary however to supplement device classes with a possibility to configure devices at runtime. This is achieved by the **resource database**. Resources are identified by an ASCII string. They are associated with devices via the device name. Resources are implemented in the device class. A well designed device class will define all device dependancies (e.g. hardware addresses, constants, minimum and maximum values etc. etc.) as resources. At runtime the device class will interrogate the database for the list of resources associated with each device it must serve. This is done during device creation and/or initialisation time. Resources allow device classes to be completely general and flexible.

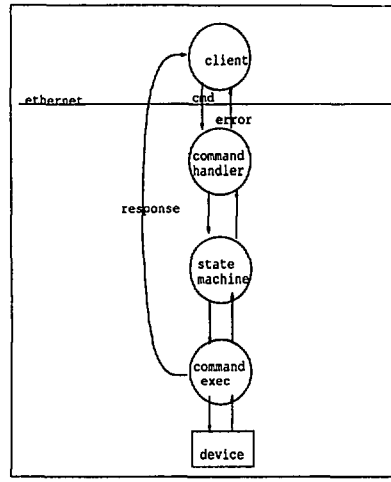

Figure 1: Command execution in the Device Server Model

At the ESRF the resources are stored offline in **Oracle** where they can be accessed using an **SQL*Forms** application. Online access is provided by a database server which interrogates a memory resident database (**RTDB**).

3.8 Commands

Each device class defines and implements a list of **commands**. The commands are the applications **dials** and **knobs** of the device. Commands (unlike methods) can not be directly inherited from superclasses. They have a fixed calling syntax – consisting of one input argument and one output argument. Arguments can be complicated structures. Commands can vary from simple **On/Off** type actions to complicated sequences which involve a large number of steps. Defining commands is the task of the class implementor and his client. The list of commands to be implemented depends on the definition of the device and the implementation of the class. Because all commands are executed synchronously care has to be taken that the execution of a command does not take longer than the maximum allowed time.

All commands are theoretically executable from a remote client across the network. It is possible however to define a restricted usage of certain (or all) commands. This ensures security for sensitive devices. Commands are executed using the application programmers interface. Before a command gets called the root class checks to see

wether the argument parameters have been correctly specified. The state machine is also called to see wether the desired command can be executed in the present state. This is implemented in the **command_handler** method of the root class (see *figure 1*).

Global standard commands have been defined which are implemented in every device class e.g. **DevState** to read the device state. It is possible to define subsets of standard commands for devices belonging to the same superclass which are to be implemented in all subclass of that *superclass*.

3.9 Local Access

Not all clients of device access and control are network clients. It is sometimes necessary to use the device class locally. The notion of local access is completely compatible with the device server model due to the adoption of **OOP** techniques. Device classes can be used locally (as opposed to remotely) in a number of ways –

- as a superclass,
- as a subclass,
- as a local sub–object within a class.
- or as a local object in an application,

This allows applications which have to run close to the hardware because of performance or hardware constraints, to profit from existing device classes.

3.10 Network Access

Network access is implemented in the device server model in the root class. This is achieved by a **remote procedure call**. The *de facto* industry standard from **SUN** – the **NFS/RPC** has been chosen because of its wide availability.

Two types of network access are provided –

- *device*,
- *administrator*.

For this reason there are two *api's*. The device api is described below. The administrator's api supports a kind of meta-control to device servers. Using the administrator's interface various kinds of useful information can be obtained about the device server e.g. the devices being served, and the number of clients per device. The administrators interface also supports a number of commands e.g. shutting down the server, restarting the server and reconfiguring the server.

Parameters passed between clients and server have to be converted to network format (for NFS/RPC this is **XDR** format) and back again. These tasks are known respectively as *serialising* and *deserialising*. A central library of *serialising* and deserialising routines is maintained as part of the root class. This way device server programmers do not have to learn how to serialise and deserialise data.

3.11 The Application Programmers Interface

A device server client accesses devices using the application *programmer's interface* (*api*). In order to improve performance the device server api is based on the *file* paradigm. The file paradigm consists of **opening** the file, **reading** and/or **writing** to the file and then **closing** the file. The device server api paradigm consists of –

1. **importing** the device using

```
dev_import (name,ds_handle,access,error)
char       *name;
devserver  *ds_handle;
long       access;
long       *error;
```

2. **putting** and/or **getting** commands to the device using

```
dev_putget (ds_handle,cmd,argin,intype,
            argout,outtype,error)
devserver  ds_handle;
short      cmd;
DevArgument *argin;
DevType    intype;
DevArgument *argout;
DevType    outtype;
long       *error;
```

3. **freeing** the device using

```
dev_free (ds_handle,error)
devserver ds_handle;
long      *error;
```

Using these three calls a client can execute any command on a device. The client uses a local procedure call to access these functions. The local call is then converted into a network call by the *remote procedure call* mechanism (ref. [5]). Each call is a **blocking synchronous** call. This means that the client **waits** until the call returns before continuing. If the server doesn't respond or the remote machine is down a **timeout** will ocurr.

Variations of these three calls supplement the basic device server api. For example a **vector** version of the above calls exists which takes a list of devices and a list of commands to be executed, thereby reducing the network overhead incurred by the rpc. Work is also continuing on an **asynchronous** version of the **dev_putget()** call which will dispatch the command and then return immediately. The response will be queued and returned to the client when it is ready to receive it.

4 Example

The device server model has been successfully used at the ESRF to solve the problem of device access and device

control for the entire injection/extraction part of the machine. A typical example of a device class is the Powersupply class.

4.1 The Powersupply

At the ESRF one of the most common device types is the powersupply. There are over 300 powersupplies from over 10 different suppliers. Hardware interfacing is either via a serial line or a G64 bus. Each supplier uses a different register description or protocol.

Using the device server model it has been possible to hide these hardware software differences between the different powersupplies. Applications see a generic powersupply which behaves identically for all powersupplies. The generic powersupply is a device in the device server model. All powersupplies belong to the same superclass – the **PowerSupplyClass**.

The PowerSupplyClass defines a partial structure for each PowerSupply device/object. Every subclass of the PowerSupplyClass uses the fields defined in the Powersupply partial structure. This way all powersupply classes have the same definitions and are easier to implement, understand and maintain.

The PowerSupplyClass is what is known as a container class i.e. it is never instantiated – its job is

- to provide a framework within which to implement subclasses,
- serve as a receptacle for common powersupply related methods.

Each new powersupply class is implemented as a subclass of the PowerSupplyClass. The convention has been adopted to name the powersupply classes after the supplier. *Figure 2* shows a synoptic of the powersupply class hierarchy for some of the powersupplies involved in the injection/extraction process.

In order to standardise the behaviour of all powersupplies a set of commands have been defined which are implemented in every powersupply class. These are –

- **DevOff**, switches the powersupply off.
- **DevOn**, switches the powersupply on.
- **DevReset**, resets the powersupply after a fault condition has ocurred.
- **DevState**, returns the state of the powersupply as a short integer.
- **DevStatus**, returns the state of the powersupply as an ASCII string.
- **DevSetValue**, sets the principal setpoint (current) to the specified value.
- **DevReadValue**, returns the last set value and the latest read value.

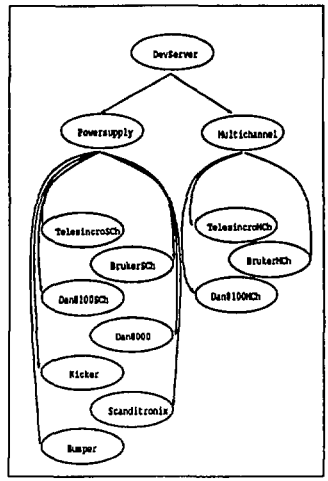

Figure 2: Class Structure for Injection/Extraction Powersupplies at the ESRF

- **DevUpdate**, returns the state, set value and read value.

For more complex powersupplies additional commands can be added to this list e.g. **DevStandby**.

The approach to define an generic device of type powersupply has proved very powerful for applications. The application programmers can develop their applications completely independantly of the device class. This way both programs can be developed simultaneously and be ready at the same time.

5 The OOP Experience

OOP techniques were chosen for the device server model partly because the problem (a general device access system) falls in the scope of problems which can be solved by OOP techniques; partly as an experiment in OOP to see what it *really* implies.

From the experience with the device server model the following advantages could be identified :

- the possibility to **inherit** code from existing classes,
- the **logical structure** it imposes on classes,
- the fact that it reduces the **coupling** between code to a minimum.

Not all experience with **OOP** has been positive however. One of the main drawbacks with **OOP** is the time it takes beginners to learn. Our experience has shown us that even though **modelling** the world in terms of objects might come naturally to many people **programming** with classes does not. It takes longer for programmers having no experience with **OOP** techniques to become productive than it takes programmers to become productive in a project using traditional (**procedural-based**) techniques.

6 Standardisation

Device servers can be regarded as a **new generation of device drivers**. They provide device independant access within a distributed computing environment. The device server model could be used as a basis for a standard way of solving device access in a distributed environment. The main areas that requires standardisation are the **network protocol** and the **class hierarchies**.

The present implementation of the device servers has defined a device access as consisting of a **command** with **one input argument and one output argument**. In order to arrive at a standard protocol the **commands** and the **data types** to be supported by the standard would need to be defined. Doing this could lead to a a standard access mechanism for devices in a distributed environment (much in the same way that the **X11 protocol** is a standard in graphics programming in a distributed environment).

Standardisation work is also required for class hierarchies. A standard **superclass** should be defined for each of the basic device types. The standard fields to be used by each member of a certain superclass will thereby be defined. A minimum set of commands to be implemented by each member of the same superclass need to be defined as well. This will ensure consistent behaviour of all devices belonging to the same superclass and maximum reusage of code.

7 Conclusion

In this paper a model, called the **device server model**, has been presented for solving the problem of **device access and control** faced by all control systems. **Object Oriented Programming** techniques were used to achieve a powerful yet flexible solution. The model provides a solution to the problem which hides **device dependancies**. It defines a software framework which has to be respected by implementors of device classes – this is very useful for developing **groupware**. The decision to implement **remote access** in the root class means that device servers can be easily integrated in a distributed control system.

A lot of the advantages and features of the device server model are due to the adoption of **OOP** techniques. The main conclusion that can be drawn from this paper is that

1. the device access and control problem is adapted to being solved with **OOP** techniques,
2. **OOP** techniques offer a distinct advantage over traditional programming techniques for solving the device access problem.

8 Acknowledgements

The authors would like to express their thanks to the first generation of device server programmers (who programmed valiantly on even before *the doc existed*) – D.Carron, P.Mäkijärvi, T.Mettälä, C.Penel, G.Pepellin, M.Peru, B.Regad, B.Scaringella, M.Schofield, E.Taurel, R.Wilcke and H.Witsch. As the first generation of device server programmers they tested the device server model on a wide variety of devices and provided valuable feedback on the validity of the device server model. The device servers they wrote were used to build the ESRF machine control system and the first beamline control system.

References

[1] J.L. Laclare, "Overview of the European Synchrotron Light Source" in *IEEE Particle Accelerator Conference*, Washington, D.C., March 1987, pp. 417-421.

[2] "Off the MAP" in *Scientific American*, August 1991, pg. 100.

[3] P.J.Asente and R.R.Swick, *X Window System Toolkit*, Digital Press, 1990.

[4] K.E.Gorlen, S.M.Orolow and P.S.Plexico, *Data Abstraction and Object Oriented Programming in C++*, John Wiley & Sons, 1990.

[5] A.D.Birell and B.J.Nelson, "Implementing Remote Procedure Calls" in *ACM Transactions on Computer Systems* 2(1), February 1987.