INTERPROCESS COMMUNICATION AND X/MOTIF-BASED USER INTERFACE
IN THE DESIGN OF AN OBJECT ORIENTED REAL-TIME APPLICATION
OR
"OBJECTS AT SEA"

Victor Nicolaysen, Ingvar Dokken and Frode Langset
Norwegian Defence Research Establishment
Division for Electronics
P.O. Box 25
N–2007 Kjeller
Norway

Abstract

The paper presents some of the design experiences obtained from the implementation of a prototype of an Electronic Chart Display and Information System. NDRE has designed and implemented a prototype as a part of an integrated bridge for high speed vessels. The paper discuss some general problems concerning classes for interprocess communication. We are addressing the problem of transferring data structures of objects and the problem of transferring single objects between different processes. The implementation of Simulatype classes using X/Motif to create the man machine interface is briefly described.

1 INTRODUCTION

Fast passenger ferries have gone through an extensive improvement the last years, both regarding speed, reliability and passenger comfort, and Norway has been in a leading position in this field. The next generation of ferries will cruise at about 50 knots with an even higher demand for safety, reliability, economics and comfort. The bridge design of existing ferries does not meet the requirements for these desired qualities. One of the major disadvantages of existing bridge designs is the lack of a good man–machine interface. Most bridge designs give an impression of being a mechanical integration of separate modules rather than a unified system.

NDRE has in cooperation with Kværner Eureka A.S. designed and implemented a prototype of an Electronic Chart Display and Information System (ECDIS). The ECDIS consists of a dynamic map and a communication module. It is a navigational support system where the captain selects the composition of the desired information. The system is continuously showing the crafts position in a dynamic naval chart (i.e a map) and displays available sensor information like satellite navigation (GPS – Global Positioning System), speed log, radar, gyro compass, etc. The software design of this system had to take into consideration the modelling of both the static and dynamic elements of the chart, parts of the physical environment, all the different sensors and
the man–machine interface. The process of modelling all these elements raised questions concerning how to communicate between Unix–processes and how to incorporate the Simula-type class with X/Motif.

Figure 1.1 Integrated bridge

2 DESCRIPTION OF THE SYSTEM

2.1 General overview

The system is far too complex to be implemented in a single process. The different tasks of the system are easily identified and therefore located in separate processes. This leaves the distribution of computer resources to the operating system.

The system is today made up by a handful of processes. Among these are processes for communication between operator and system, Man–Machine–Interface (MMI) and for displaying the chart (Chart Display). There is a process called the sensor control process (Sensor Control) that receives the messages sent from the sensors and there is one for logging the last hours of the craft's journey (Record). There are also processes for detecting input signals, like pointer movements and pushbutton activations. The Chart Manager ensures that the Chart Display process displays the correct geographical area, and there is a process which communicates with the radar (Radar Control). The system does not have a dedicated synchronizing mechanism.

Figure 2.1 shows an imagined interior of the system. The real world is shown through the windows (i.e the ECDIS and the environment) and the inside illustrates the different processes.
The system is developed on a standard, high-performance graphical UNIX workstation with real-time extensions using C++, X/MOTIF and a graphic library (GL).

2.2 Real world objects and pure software objects

The different processes in the system are based on an extensive use of objects. Objects, which are instances of classes modelling aspects of the real world or pure software aspects as network behavior or memory allocation.

The design is focused on modelling the world seen from the outside, i.e. the user’s perspective, and inward towards the pure software objects, i.e. abstract entities only meaningful for the programmer. This strategy ended up in different layers of class hierarchies. At the top level we have entities that seem meaningful for the user and deeper down we find entities only useful for the application programmer.

The system contains objects like ship, craft, gps-receiver, gyro compass and speed-log. These are typical real world objects i.e. the software objects (instances of the classes) that have physical counterparts in the real world. We have tried to implement the system in such a way that where it is meaningful to get a service from a real world object it should also exist a class that provide the same service. We do also believe that this strategy will make the maintenance task
easier, because there is a mapping between the intuitive real world object and the source code concerning the services such a device is providing.

Other objects don’t have real world counterparts, i.e. for example shared memory and semaphores. These are parts of the layers further away from the user of the system. These objects are in a way modelling the "real" world for the application programmer. Typically the real world classes are internally using pure software objects. This means that classes at the upper level consists of instances of the lower level classes. One might say that these higher levels consist of lower level classes as building blocks.

3 BUILDING BLOCKS

3.1 General comments

The system contains sets of classes, but most of these classes are subclasses of more abstract classes, which themselves are never created during runtime, but serve as containers for the general concepts concerning all the subclasses. We are in the following briefly showing some of these class hierarchies in order to end up in the interprocess domain.

We have aimed at not making the processes too complicated. The complexity are mainly hidden within the classes.
3.2 Chart–object classes

A chart–object is a member of the set of different objects visible on the screen. A chart–object could be the own craft, a radar target, a lighthouse, a buoy, a chart symbol, a text or simply a geographical point. These objects have real world counterparts but they also consists of a composition of other objects. The own craft class for example does contain sensors like gps–receiver, gyro compass and speed–log. Like in the sensor class case the class Chart–object does contain the common parts of all chart–objects and does in fact hide the implementation details for the application programmer on the next level.

![Chart–object class hierarchy](image)

Figure 3.1 Chart–object class hierarchy

The instances of the subclasses of the chart–object are objects that should be sent around in the system as "messages". This is a kind of message that has a specific destination determined by the sender. We use class Message to accomplish this task. Class Message is using a Unix system V message queue to send a message of x bytes. At this stage does one problem occur i.e. how is it possible to transfer a correct table of virtual procedures and how does the receiving process know which type of object that has arrived? These problems will be discussed in chapter 4. In C++ each instance of a class contains a table of the addresses where the virtual procedures are to be found (1) (5). This table refers to the process’ own virtual address space, and will therefore be useless in another process.

3.3 Sensor classes

The notion of a sensor in the system is any real world object providing information about the exterior i.e. the crafts position in a defined earth based reference system, the crafts heading in the same system, the crafts speed in an appropriate reference system, the temperature conditions etc.

The sensor messages are provided through networks and communication protocols. But for the application programmer who really doesn’t need to know which type of communication there have been used this information should be hidden. This is done by hiding the communication for all sensors in a common base class called class Sensor. All the different real world sensors like
the gps-receiver and the gyro-compass are implemented as subclasses (derived classes) to class Sensor.

![Sensor class hierarchy]

**Figure 3.2** Sensor class hierarchy

### 3.4 Interprocess classes

The interprocess (ipc) classes are mentioned earlier as tools to provide the necessary communication between processes. The interprocess class hierarchy consists of a baseclass ipc-object, which is an abstract class that never will exist during run time. It is a common base for the different underlying subclasses. Three of these classes are; class Shared memory, class Message and class Semaphore (8). These three classes are encapsulating and abstracting most of the details in the Unix system V interprocess facilities (3), and together can they be used as building blocks to create more robust entities.

![Interprocess class hierarchy]

**Figure 3.3** Interprocess class hierarchy

The class semaphore is an implementation of a named set of semaphores (5). Class Message is an implementation of a named message queue, and class Shared memory gives a named shared memory segment. These rather general classes can then be used to build access controlled structures like monitors (5) (7), where the access to a shared memory segment are controlled by the means of atomic actions on a set of semaphores.

### 4 INTERPROCESS FACILITIES AND PROBLEMS

Unix system V provides shared memory, message queues and semaphores to accomplish communication between processes (3). Our problem was to transfer data structures of objects linked
together from one process to possibly many others. The need for this facility was based on the transfer of large data structures containing geographical and navigational information in the system. We implemented three basic classes to accomplish this task. During the design of these classes two major problems were addressed:

How to transfer data structures between different processes?

How to transfer objects between processes without getting error in the internal virtual table?

The essence of the first of these two problems is visualized in the following figure.

![Diagram showing processes attaching at different virtual addresses (DD00) or attaching at the same virtual address (CCFF)](image)

*Figure 4.1 Processes attaching at different virtual addresses (DD00) or attaching at the same virtual address (CCFF)*

It is mandatory that the receiving processes use the same virtual address for the shared memory segment as the sending process. Otherwise it is not possible to transfer meaningful pointers or addresses, they will all refer to the address space of the sending (creating) process instead of the receiving process. Figure 4.1 is showing two processes which are using either two different virtual addresses or the same virtual address in order to refer to the same shared memory segment.

We have implemented the class Shared memory by a "creator decide"—principle, i.e. the first instance of the class that connects to one particular shared memory segment decides the attach address to be used if the segment is meant to store pointers or address—dependent information. This attach address is then stored in the shared memory segment's info-header. All other processes needing address—dependent information will try to attach the shared memory segment at the address stored in this info—header.
The shared memory segments are now independent of each other. All necessary information concerning a shared memory segment is stored within the segment, but on the other hand is it not possible to create and attach shared memory segments in a random sequence. The problem arise when one process already has created a shared memory segment using the attach address X and another process has created yet another different shared memory segment using the same attach address X. When either of these to processes now wants to attach the others shared memory segment an error occurs, i.e. they are able to attach but not using the address X. One solution is to let a process initially create all shared segments used in the system before the "working"—processes starts, processes which now is bounded to use the no—conflicting addresses created by the dummy—start—process. This is obviously not a good procedure if the purpose is to make a general tool, but for our system is it sufficient. One more general solution (6) (8) is to implement a centralized distributor of shared memory attach addresses. This could be provided either by a special purpose process or simply a dedicated shared memory segment containing the addresses and the maximum sizes of the segments. Yet another possible solution for single computer applications is to put all the different processes' memory into one single shared memory segment, but this requires different access permissions to the different parts of the segment.

The second problem concerned the object's virtual table. The problem was that the table of addresses of virtual procedures for the object referred to the creating process' address space. We managed this problem by introducing a receiving procedure in the baseclass that actually created an object of the correct type and then initialized it with the data from the message queue. The paradox in this solution is that the baseclass needed information of its own subclasses, namely their existence.

5 HOW TO LEAD MOTIF INTO AN OBJECT-ORIENTED STRUCTURE OFFERED BY C++

One major problem concerning object oriented system design is the lack of good object oriented design tools and libraries. Motif and the X-window system do not have a good interface to the Simula-type class definitions offered by C++. This chapter discuss a simplified design example consisting of a pushbutton (visualized on a screen) and a text window that is visualized when the pushbutton is activated. The textwindow is updated with dynamic information.
Motif under X has no C++ like inheritance, i.e. there is no way to inherit from a Motif widget into a C++ class structure (1) (4). There is on the other hand a kind of polymorphism through 'callbacks' (pointers to functions). A list of "callbacks" together with a list of user defined data, is passed to the constructing procedure when constructing a motif widget (i.e. a push button, a window etc). The "callback"-list is then made a part of the widget's resources. When an event appears (i.e. a time has passed or the cursor is pointing to a visualized widget on the screen), the X-eventhandler looks up these lists for the widgets concerned and calls the functions therein. By inserting a C++ class member function into this 'callback'-list the widget becomes a part of the class and the class structure construct the widget (or a set of widgets) through the class constructor. The user of the class do not need to supply any mechanisms between the widgets themselves and the class. Polymorphism and inheritance are now conserved.

Since Motif and X is written in C is there a problem with passing pointers to class member functions (1) (2). This is basically because the class memberfunctions references its object by a hidden pointer to the object. The X eventhandler and library functions do not know about this pointer and even if it can call a class member function, it cannot refer to the object itself and therefore do not know about other class members. An easy way to overcome this problem is by implementing 'pseudo memberfunctions'.

A 'pseudo memberfunction' is a function which has a signature known by Motif and X, which is compiled in C++, and which knows about its object through a data structure supported every time it is called. The 'pseudo memberfunction' calls the 'real' class memberfunction in the object referred to by the supported data structure. The data structure also contains any argument that should be passed to the class memberfunction and also holds space for returned values. To keep track of the names of the 'pseudo members' a naming convention containing the 'real' name of the class memberfunction and the name of the class itself should be applied. The call from X to this 'pseudo member' looks like a normal callback:

```c
*(callback[n].callback)(aWidget , callback[n].closure , AvoidPointer);
```

The 'pseudo member' is given to Motif and X through a callback list (implemented by Motif and X):

```c
className_MemberName_ClientData.object_pointer = &TheObjectItself;
callback[n].callback = ClassName_MemberName_Callback; //The pseudo member
callback[n].closure = ClassName_MemberName_ClientData_Object; //Data structure
```

The implementation of the classes and their members may look like this:

```c
class ClassName
{
    public:
        void MemberName();
    } TheObjectItself;
```
Figure 5.1  An example with a Motif Pushbutton Widget and a Motif Text Widget

ClassName::MemberName()
{
    // Do something
}

class ClassName_MemeberName_ClientData
{
    public:
    ClassName* object_pointer;
} ClassName_MemberName_ClientData_Object;
void ClassName::MemberName_Callback(Widget w, caddr_t client_data, caddr_t call_data)
{
    ((ClassName*)client_data)->object_pointer->MemberName();
}

6 CONCLUDING REMARKS

We designed the ECDIS to consist of a set of class hierarchies. The hierarchies at the top level modelled the real world objects and the lower hierarchies in the system were pure software objects. We do hope that this approach will improve the understandability of the system for maintenance–personnel.

The interprocess problem of transferring address dependent information between processes lead us to a solution where the shared memory segments themselves contained all the necessary information of their appropriate attach addresses i.e each segment is independent of the others. On the other hand there is a set of solutions where a centralized device (i.e. process, table etc) distribute the attach addresses. These solutions could be extended beyond the scope of an application. The distributing device could distribute addresses for all applications within the computer or even within a network of computers.

The problem of transferring the table of virtual procedures was solved by avoiding the transfer, simply by creating the object within the receiving process. But it would have been easier if the creation of an instance of a class could be done by passing a kind of class identifier, which could be received together with the objects "state"–information. C++ does not offer such services.

The problem of designing classes modelling the X/Motif–based user interface were solved by introducing a "pseudo"–member function to the C++ class.

A final remark is that we feel that some of the work done with the interprocess facilities and the user interfaces could have been avoided if Object–Oriented DataBases (OODB) and X/Motif class libraries were commercially available. This is meant as an encouraging remark to those who work in those areas.

REFERENCES

(4) OSF/Motif: Style Guide. Prentice Hall, 1991


(7) A Lister: Fundamentals of operating systems, Macmillan, 1984


(9) Proceedings ECOOP'91, Springer-Verlag 1991