Development of an Object-Oriented Framework for Vessel Control Systems

ESPRIT III/ESSI/DOVER

Per Dagermo
EC-gruppen
Box 14
S-376 00 SVÄNGSTA
SWEDEN
E-mail: per.dagermo@ecgrupp.postnet.se
E-mail: per.dagermo@ec.se

Jonas Knutsson
Karlskronavarvet AB
Electronic Design
S-371 82 KARLSKRONA
SWEDEN
E-mail: postmaster@karlskronavarvet.se

Abstract

The DOVER project has developed an object-oriented framework for the vessel control system domain. The framework is to be used when developing new control systems for manoeuvring a vessel. By developing and using the framework a substantial increase of the level of reuse has been obtained. The bulk of the framework is possible to use in most kind of control systems, where you have input data that is to be used for calculating output data.

This report describes the problem domain, the project, the framework and the experiences gained from the project. In the framework, we have had great use of and help from the Design Pattern concept [Gamma95].

We describe the patterns that we have used and how we have adapted them to this particular problem. The most important pattern in the design has been the Observer pattern [Gamma95], which has made it possible to have the loose coupling needed for the large scale reuse in this domain.

1: Introduction

The DOVER project (Development of an Object-Oriented Framework for Vessel Control Systems) is an ESPRIT III/ESSI project, #10496, which means that it is financed by the European Commission. The project was conducted by Karlskronavarvet, a shipyard in Sweden, subcontracting EC-gruppen for training and consultancy.

The overall objectives with the DOVER project was to study what advantages of introducing development for large scale reuse, i.e. object-oriented framework development. When the design of the framework started, all features from older systems had to be included and it should be possible to extend them and add new features to the framework.

The base theory in the design of the framework is a continuos computation of values. A value in the framework can be either an *InputValue* or a *ComputedValue*, the ComputedValues make themselves dependent to specific InputValues and when any of the InputValues change a re-computation is performed. The framework for vessel control systems includes:

- A set of design objects which are used to develop a new vessel control system
- An execution model
- Distribution mechanisms
- Support for configuration files

Beside the framework, the following results exist:

- Demonstrators/simulators running on MS-Windows, based on the framework
- Framework documentation and a development handbook

1.1: Project Information

The DOVER project started in July 1994 and it was ended in October 1995, with a duration of 16 months. There have been 3 to 5 project members at Karlskronavarvet.

The project was followed up with the DOOFT project, which stands for Dissemination of Object-Oriented Framework Technology, ESSI project #10495. In this project we took upon us to share information about object-oriented framework development, patterns and experiences from the project. The DOOFT project consisted of 5 one-day seminars, held in 5 different European countries, and the preparation of two reports to be spread in the software developing community.

1.2: Project Execution

The main phases in the DOVER project can be seen in figure 1.1. We initially planned for a strict waterfall type of execution, but we soon realised that we had to iterate between the phases.

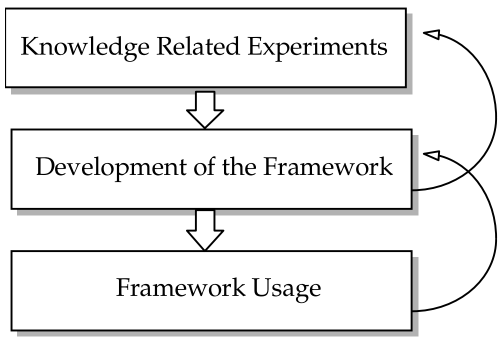

Figure 1.1: Project phases.

- The first phase was Knowledge Related Experiments. This phase was composed of a number of well defined activities of experimental characteristics, education in object-oriented theories and collecting knowledge concerning vessel control systems.
- The second phase was the Development of the Framework where the actual development was accomplished.
- The third phase was the Framework usage where the framework was evaluated and refined by developing applications.

An evaluation of the framework will be done two years after the project has been completed, i.e. after the framework has been used and necessary maintenance and changes has been done to it.

1.3: Organisations Involved in the Project

Karlskronavarvet AB is a shipyard in Karlskrona, south-east of Sweden. At Karlskronavarvet there are about 850 employees, 50 work in the electronics and software engineering department. The main business area is constructing and building naval vessels for the navy in different countries.

EC-gruppen has been a partner (subcontractor) in the project. They have their main office in Svängsta, Sweden. They focus on Consulting, Training and Research in information technology. They are about 60 employees and have experiences from other EU sponsored research projects within the ESPRIT programme.

Lennart Ohlsson, Utilia Consult, has been working in the project as mentor and external expert on design patterns and framework design.

1.4: Expected Benefits

If the project could increase the level of reuse Karlskronavarvet would gain the following benefits:

- Reduced software development cost
- Shorter software development time
- Better estimation of development time
- Higher quality

1.5: Encountered Problems

When we started this project the general opinion was that the change to object-oriented technology would go smoothly, since we already knew structured analysis/design and the programming language C, but this was not the case. The learning process took much longer time than we expected. The object-oriented analysis and design seemed to take less time to learn and accept, than object-oriented programming in C++.

The available text books on object-orientation, gave relatively little help on how to make good designs. Their main contribution was on notation and terminology. There were few publicly documented projects of this kind and little written about object-oriented design and programming.

1.6: Project Results

- A lot of the project time has been used for training and introducing object-oriented techniques. This has resulted in a staff of competent object-oriented designers and programmers.
- A framework for vessel control systems, used for developing two prototypes the Microsoft Windows platform.
- ”Technical Overview”, a document which describes our technical background and also shows the design principle in our framework.
- ”Development Handbook” , a document which is meant to be used when using the framework. It can be seen as a ”cook-book” in how to use the Framework.
- ”Experience Report”, delivered to the EU commission, explaining the advantages and problems we have had in the DOVER project.

2: Technical Background

2.1: Architecture-Based Software Development Process

The object-oriented approach supports reuse. However, reuse has been intended mostly at the component level. This has generated the growth of libraries of reusable classes. But, more recently it has been observed that a higher level of reuse is achievable by reusing entire architectures instead of just components. To offer a method and an understanding for reasoning in terms of architecture the concept of *Object-Oriented Design Patterns and Frameworks* has been introduced.

2.2: Patterns in Object-Oriented Design

It is generally accepted that designers always have been driven by (personal) "patterns" when making choices. The architect Christopher Alexander has in [Alex77] formalised this idea. Alexander suggests that ”each pattern describes a problem which occurs over and over again in our environment, and they describe the core solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”.

The concept of *design patterns* has since the early 90’s been considered in object-oriented literature as a possible way of coping with architecture reuse. Here are some descriptions of the concept. A design pattern:

- is a simple and elegant solution to specific problems in object-oriented software design.
- captures solutions that have developed and evolved over time - "The quality without a name".
- names, motivates and explains a general design that addresses a recurring design problem in object-oriented software

But, a design pattern is *not* a design - it has to be ”instantiated”, i.e. you have to:

- evaluate trade-offs and consequences
- make design and implementation decisions
- implement and combine with other patterns

Based on the ideas of Alexander, Gamma et. al. [Gamma95] have written a book, which is both describing the concept and a catalogue of design patterns. For reasons of generality and reusability, the patterns are fairly application and language independent.

2.3: Object-Oriented Frameworks

Object-oriented frameworks represent the scaling up of the fundamental principle of object-oriented programming is that, due to the features of inheritance and dynamic binding, a new class can be implemented by providing only what is different in this class compared to one which already exists. With object-oriented frameworks, the same principle is applied to whole applications or subsystems, so that the effort to develop a new application is proportional to the difference in functionality between the particular application and that in the framework. The goal is "Large scale reuse", i.e. to write as little new software as possible when developing new applications. It is obviously more powerful to reuse the design and code rather than only lower-level code components.

In contrast to traditional approaches to software reuse, which are built from libraries containing a large number of small building blocks, object-oriented frameworks allow the highest common abstraction level among a number of similar products or applications to be captured in terms of general concepts and structures. This creates a generic design that can be instantiated for each existing product as well as for products to be developed and marketed in the future. Frameworks are therefore ideally suited for capturing the commonalities in a product family. Each product is an instantiation of the framework, where the amount of unique code is proportional only to the amount of specific features in that product and not to its total complexity. There are, for example, object-oriented frameworks for window-based user interfaces, such as Borland OWL and Microsoft Foundation Classes.

An object-oriented framework consists of a set of collaborating classes (both concrete and abstract classes). To develop a new application, use and add new classes by inheriting from classes in the framework. You don't change the framework when developing a new application, instead you extend it with what is not general to all applications in the domain - "modification by extension". Figure 2.1 shows the different levels of reuse.

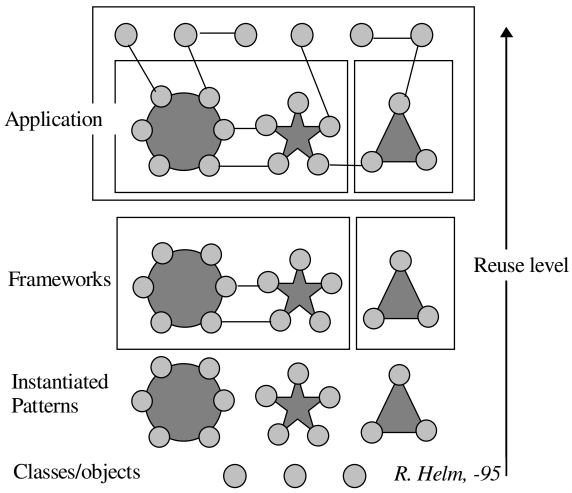

Figure 2.1: Reuse on different levels.

2.3.1: Class Libraries vs. Frameworks

1. A Framework provides an integrated set of domain-specific functionality.

- Components in a framework typically address a particular domain (such as business data processing, telecommunications, graphical user interfaces, databases, and distributed communication systems).
- Class library components are mostly domain-independent (such as strings, complex numbers, dynamic arrays, and bitsets). Since a framework is more tightly integrated with a particular domain, the scope of reuse is significantly larger than using class libraries.

2. Frameworks exhibit an "inversion of control" at run-time.

- Designs based on class libraries traditionally placed the application in control. The application programmer was in charge of designing the flow of control through the application.
- In contrast, designs based on frameworks place the framework in control. Developers that use a framework write code that can be called by the framework, instead of writing code that calls a library. Thus, applications reuse the flow of control and the application architecture that the framework provides.

3. A framework is a "semi-complete" application.

- Programmers using a framework(s) create complete applications by inheriting and instantiating parameterised framework components (rather than simply invoking methods provided in a class library). Inheritance enables the features of framework base classes to be selectively shared by subclasses. Only those methods that need custom application-specific functionality must be overridden.

2.4: Polymorphism and Dynamic Binding - the Basis for Frameworks

The ability in object-orientation to hide different implementations behind the same name is called polymorphism. An often used example to explain what it is and how it works, is Shapes in a drawing tool. The software that handle shapes of different kinds (rectangle, square, line, text, etc.), can be written so that all shape classes are treated equally, i.e. they have the same interface. Shape is the base class, defining the interface. All subclasses must implement their own specific behaviour, most notably how to draw themselves. To draw all objects in a set, you just call Draw() for all object, without having to know the actual type of shape your dealing with.

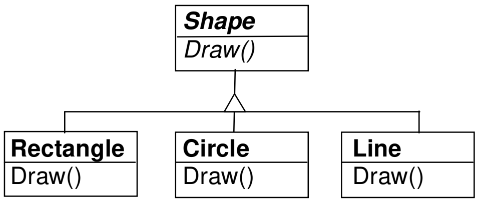

Figure 2.2: The Shape hierarchy.

The Shape class is an abstract class (shown by *italics* font), meaning that you should not create objects from it, instead, it defines the interface for subclasses. The interface consists partly of abstract operations (also in *italics*), which all subclasses have to implement in order to conform to the common interface.

Polymorphism greatly simplifies communication among objects. You can use the same name with many different kinds of objects to trigger the same effect without having to deal with the underlying mechanisms, which may be quite different. This works as long as they have the same base class, defining the common interface.

Dynamic binding is the way object-oriented programming languages implement polymorphism. In C++, a function is dynamically bound if the keyword `virtual` is used. Polymorphism and dynamic binding is the basis for object-oriented frameworks, as we will show with a ”very small framework” example, see figure 2.3.

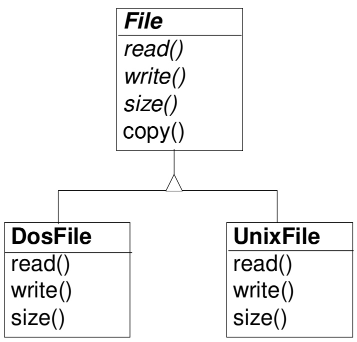

Figure 2.3: A very small framework for files.

```
class File {
public:
  virtual void read(char*)=0;
  virtual void write(char*)=0;
  virtual void size()=0;
  void copy(File);
};
class DosFile: public File {
public:
  virtual void read(char*);
  virtual void write(char*);
  virtual void size();
}; // and the same for UnixFile
File::copy( File* aFile ) {
  char buffer[ BlockSize ];
  for ( int i=0; i<=this->Size(); i++ ) {
    this->read( buffer ); // dynamic binding!
    aFile->write( buffer );// dynamic binding!
  }
}
// Somewhere in the application:
File*  aDosFile  = new DosFile;
File*  aUnixFile = new UnixFile;
// copy from aDosFile to aUnixFile.
aDosFile->copy( aUnixFile );
```

Example 2.1: Code for the file ”framework”.

Imagine that the File class is in the framework, and that DOS_File and UNIX_File is what the user of the framework has to supply. Thanks to dynamic binding, calls go from the File::copy() operation to read() and write() operations in the subclasses. This is why calls can go in both directions, see figure 2.4.

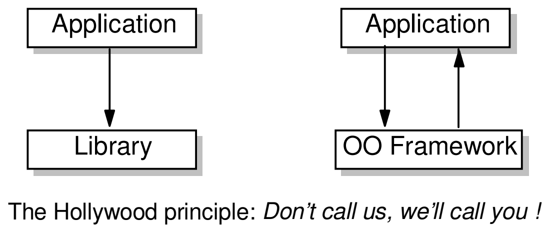

Figure 2.4: Calls are go in both directions when using frameworks.

The File example is based on a design pattern from [Gamma95], called Template Method. It is perhaps the most important design pattern for framework development.

Problem: A common behaviour in subclasses differ only in some details, and you get a lot of code duplication.

Solution: Define a skeleton of an algorithm in an abstract operation, deferring some steps to the subclasses.

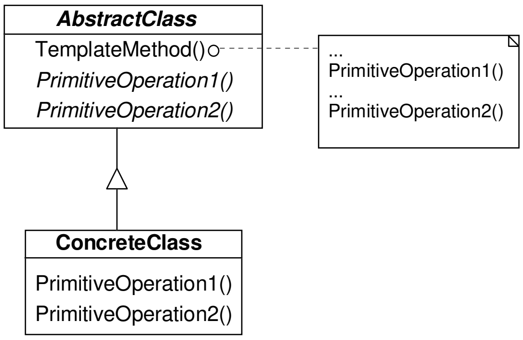

Figure 2.5: The structure of the Template Method design pattern.

2.5: Distributed Object Technology

Distributed systems are by their very nature large and complex applications. They require interaction of many individual components scattered throughout a distributed collection of hosts. Such systems are natural candidates for the object-oriented model of software development. Object-oriented technology and distributed systems technology may seem very different in scope and purpose, but they complement each other very nicely.

Independent objects can be created to manage all of the complexities of distributed processing, allowing the user to freely connect and exchange information across widely diverse hardware and software systems.

3: Domain Description

This domain description is divided into a general and a variant part. The general part describes the common characteristics of different vessel control systems, and the variant part describes those details that often differs between different vessels. The general parts are to be covered by the framework, and the variant parts must be supplied when a new system is developed.

3.1: General Aspects

This first part describes a vessel control system in general. Things like system and functional characteristics, different safety, alarming and emergency handling will be discussed.

3.1.1: System and Functional Characteristics

A Vessel Control System is the system responsible for the controlling and manoeuvring the vessel (ship). The vessel is controlled/manoeuvred by one or several co-operating officers at one or several *control stands*. A control stand is a place from where the vessel is controlled.

There are three different control stands: Wheelhouse (WH), Engine control Room (ECR) and Portable control stand (PCS). The main control position is the wheelhouse but the PCS is suitable to use when you want to be out on the deck and manoeuvre the vessel, for example when you want to make harbour. A control stand may be equipped with one or several levers, wheels, joysticks etc.

The system must perform in a well defined way under all conditions, even when something is broken. Therefore, the system consists of redundant sensors and computers.

The ECR computer provides an interface for an autopilot, GPS (Geographical Position System) and echo sounder and it also provides a Man-Machine-Interface (MMI) for the operator of the vessel. The MMI gives the operator feedback information about alarms, the speed through water, geographical position, direction the vessel moves and status of the equipment onboard the ship.

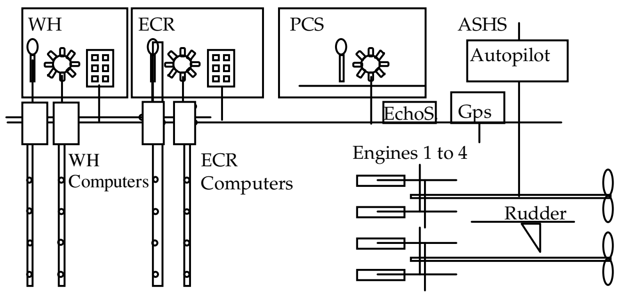

Figure 3.1: Schematic picture of a typical vessel control system.

Often the manoeuvring functions and the engine control functions are separated. The manoeuvring functions are related to navigation. The engine control functions includes start/stop and clutch in/out of the engines. Engine control is often performed by the officer responsible for machinery monitoring.

In a manoeuvring mode, the engines' speed can be controlled from one control stand and the rudder from another control stand, or from an autopilot. Start/stop functions can also be controlled from one or several control stands, not necessarily connected to the manoeuvring control stands. The configuration can be quite complex and there is no restrictions on possibilities.

3.1.2: Engine Control and Synchronisation

The operations that must be performed on the engines are start/stop, clutch in/out and setting of the RPM (Revolves Per Minute). To be able to start the engines, different requirements on other equipment must be fulfilled, e.g. the clutches have to be in specified positions.

3.1.3: Steering

The steering function is dependent on the particular rudder or propulsion system onboard the vessel. This can for example be normal rudder machinery, water jet system, bow thrusters, etc. When the steering and propulsion system is, for example, Voith Schneider Propellers, co-operating algorithms for changing the pitch and revolution of the propellers are needed.

3.1.4: Auto Pilot

The auto pilot can be an external system connected via serial link, or a more complicated navigation system controlling all propulsion parameters, including speed. The autopilot function can also be integrated in the vessel control system.

3.1.5: Safety Aspects

The safety aspects of a ship control system includes fault handling, emergency control and a redundant system architecture. Alarms detected from the controlled devices may initiate fault handling operations. Alarms from the engines may result in actions taken, e.g. reduced speed or power. The alarm information is displayed to operating personnel and/or logged for later evaluation. The best solution to fault handling is to use redundant systems, where the most healthy system always is in control.

3.2: Variant Aspects

The problem is that many things vary between different types of vessels. For example, the main mission for a special kind of vessel can put special requirements on speed, another one on manoeuvrability, etc.

3.2.1: Engine Configuration

The engine configuration often differ, e.g. the number of engines per shaft. The system must also provide algorithms for how and when the engines are clutched in and out on the propeller shafts.

3.2.2: Propulsion

The propulsion system can be ordinary propellers, waterjet or Voith Schneider propellers. This also affects the steering, it could be normal rudders or with waterjet it is the direction of the waterbeam that decide the direction. With the Voith Schneider propeller the propeller blades change their pitch depending on where in the revolution they are. This means that the control algorithms for these kinds of propulsion systems will be more complicated than when the propulsion and steering systems are separated.

3.2.3: Input and Output Signals

The system must have several digital and analogous in and output channels that are able to both read and provide data. The I/O system between the control system and the external environment can consist of both serial connections, with different protocols, and different kinds of buses e.g. VME-bus, CAN-bus or Profibus. Between different vessels the number of sensors and actuators differ and this put special requirements on the number of both digital and analogous I/O channels.

3.2.4: Hardware Dependencies

The hardware platform for existing ship control systems developed at Karlskronavarvet, has historically most been Motorola-based processor boards connected to VME-bus systems. This is not likely to be the only available solution in the future. The systems supporting the vessel control system are often based on a PC-platform.

4: Description of the Framework

4.1: Introduction

The framework provides an architecture and an adaptable class library for the application engineer to use when designing a new vessel control system. Only functionality not included in the framework design have to be added, the rest of the design and implementation effort is merely connecting building blocks in the framework.

Lennart Ohlsson, Utilia Consult, has been an important source of design ideas for the DOVER project. Parts of the text presented concerning the design is extracted from his articles.

4.2: The Value Concept

A control system essentially performs a continuos re-computation of a number of output values as the response to changes to a number of input values. The concepts which are prime candidates for classes in a control system framework is therefore simply *values*.

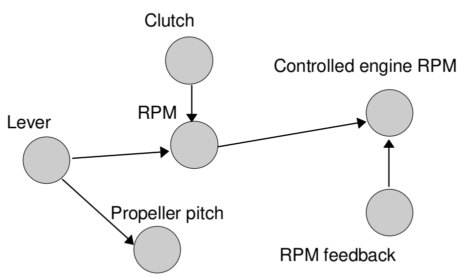

Figure 4.1: Typical values in a vessel control system. The lines show notifications of changes.

A value in the framework can either be an *input value* or a *dependent value*. A computed value can *register* itself as dependent on another values. When a value changes, it notifies all its dependent values. When a dependent value is notified, it invokes its formula, which uses `operator float()` to read the value of the observed value. The change is then propagated to its dependants, and so on.

The framework allows values to be of arbitrary types, by the use of templates. Below is the class diagram for Values, using the *Observer Design Pattern* [Gamma95], see also 6.1.

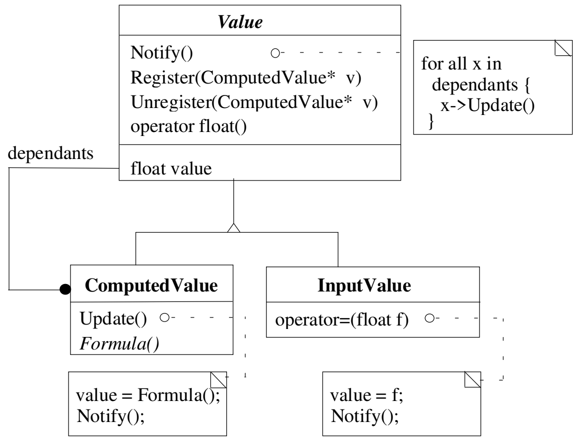

Figure 4.2: The Value design.

Note that for the InputValue class, the assignment operator is overloaded. But, for ComputedValue objects, it should not be possible to use the assignment operator. ComputedValues are only modified when the value it is dependent on change. When an InputValue object is assigned, it invokes Notify(), which calls Update() for all dependants in the list. The design permits that an object is dependant on ComputedValue objects too, which makes it possible to form whole chains of dependencies. The most important code for the classes are listed in example 4.1.

```
class Value {
public:
   void Register(ComputedValue* dependant);
   void UnRegister(ComputedValue* dependant);
   operator float() { return value; }
protected:
   Value() {}  // Abstract class
   virtual ~Value() {}
   void Notify(){
      // for all dependants
      iterator()->Update();
   }
   float value;
private:
   List<ComputedValue*> dependants;
};

class InputValue : public Value {
public:
   InputValue& operator=(float f) {
      value = f;
      Notify();
      return *this;
   }
};

class ComputedValue : public Value {
public:
   virtual void Update() {
      value = Formula();
      Notify();
   }
protected:
   virtual float Formula() = 0;
};
```

Example 4.1: Code for the Value classes.

4.5: Dependency Graphs

Dependency graphs can be used to show the dependencies between different values in the system. As an example, an InputValue object could be a lever position object and a ComputedValue object could be the RPM reference object. When the lever position changes, the RPM should also change. The arrows in the figure below define the dependency between different values. The semantics of the figure is that a change of the InputValue needs to be followed by a re-compilation of the ComputedValues.

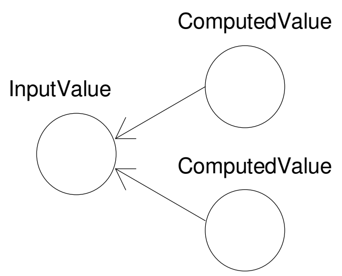

Figure 4.3: A typical dependency graph.

Figure 4.4 shows an *event trace*, describing the dynamic behaviour when objects from InputValue and ComputedValue interact.

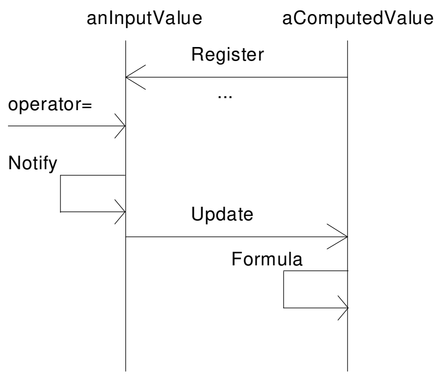

Figure 4.4: Event trace showing the sequence of messages.

The event trace (similar to *use cases*) above is interpreted as follows: There are two objects involved, one of class InputValue and the other of class ComputedValue. They each have a vertical line representing them. The time scale floats from the top to the bottom. The arrows denote messages sent. We can, for example, see that the event trace starts with a ”register” message from the ComputedValue object. The InputValue object is assigned. It calls the dependants’ Update operation which calls the Notify operation. Then the ComputedValue is re-computed with its formula.

4.3: Geometry Program Example

The following geometry program shows the principles in more detail. For clarity, the code below only shows the specific case when the type is float. There is only one InputValue object, called *theRadius*. A diameter and an area object are dependent on the radius. The circumference is dependent on the diameter, just to show the flexibility of the design.

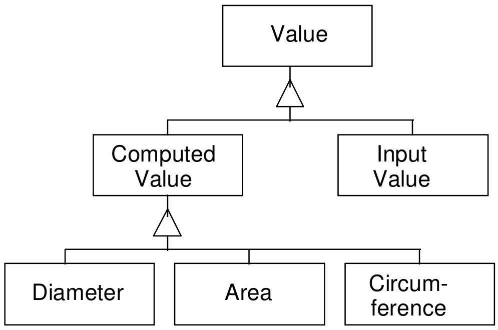

Figure 4.5: Class hierarchy in the geometry example.

When we change the radius by assignment, the change is automatically propagated to the dependent values, i.e. theDiameter and theArea. When the diameter is changed, it sees to it that the circumference is notified.

```
class Diameter : public ComputedValue {
 public:
   Diameter::Diameter ( Value& aRadius )
         : myRadius (aRadius) {
      myRadius.Register( *this );
   }
   float Formula () { return 2*myRadius; }
 protected:
   Value& myRadius;
};

void main()   {
InputValue     theRadius;
Diameter       theDiameter(theRadius);
Circumference  theCircumference(theDiameter);
Area           theArea(theRadius);

while (True) {
   float r;
   cout << "Radius: ";
   cin >> r;
   theRadius = r; // updates all values!
   cout << "Diameter: "<< theDiameter<< endl;
   cout << "Circ.:"<<theCircumference << endl;
   cout << "The Area is:" << theArea << endl;
   }
}
```

Example 4.2: Code for the geometry example.

4.4: Inheritance Tree for Values

This chapter will show the framework classes and how they inherit from the abstract base class Value in the framework. The Value class is the major base class that most of other classes inherit from. The inheritance is from left to right and classes that are indented two steps, i.e. have two classes above and to the left of itself have multiple inheritance.

```
Value
    InputValue
        InputEventValue
    ComputedValue
        PIDControlValue
        ScaledValue
        Alarm
            Difference Alarm
            Interval Alarm
        ScaledValue
        SwitchValue
    ClockManager
    Clock
    Timer
    PulseValue
    NetworkCopyValue
    NetworkSourceValue
ValueGroup
    ControlStand
    ValueGroupSwitch
    BinaryCrossBar
NetworkManager
ProgramResource
    Parameter
    Function
ResourceManager
```

Figure 4.6: The most important classes in the framework.

4.6: Execution Model

The vessel control systems developed by Karlskronavarvet have always been designed with a real-time operating system as a core. The major reason for this is that it provides for easy division of necessary activities into concurrently executing tasks. The real-time requirements of the different activities has been important design issues. For example, the task that scans for user inputs has been given one frequency and the serial link task has another, and so on with the other tasks.

But, there are often problems with synchronisation in concurrent systems with pre-emptable tasks. This is a common source of problems when certifying the real-time requirements. A low-priority task can prevent a high-priority task in several ways, e.g. when they use the same resource, such as a data structure. A solution is to not allow pre-emption of tasks, but that can lead to other undesirable effects.

4.6.1: The Time-Triggered Approach

The execution and synchronisation model used in the Dover project is based on the *time triggered approach*. When the time triggered approach is utilised, the control system observes the state of the environment at specific points in time, after which the control system decides what actions must be taken.

All aperiodic events are converted to periodic events by estimating the minimum time between changes. The input values can then be polled with different frequencies. When an input value is changed, that change will propagate through the dependent values until the chain ends. This notification scheme is shown in the following figure 4.7.

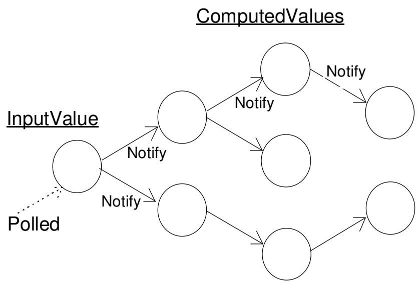

Figure 4.7: The input values are polled with the desired frequency.

In the Dover framework design the input values are updated with the current I/O port value in the "main" program, which is a never terminating while loop.

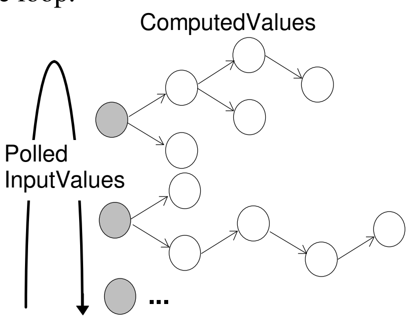

Figure 4.8: The execution model.

A very important reason to use the time triggered approach is when the system is safety critical. When a system is safety critical it must be possible to prove, or at least show, that the system behaviour is correct. When the time triggered approach is used, the dynamic behaviour of the system will be much simpler and thus much easier to test and verify compared with an event triggered system [Kopetz92], where the external or internal events trig tasks non-deterministically.

4.6.2: "Lazy and Eager" Values

Some values do not need to be notified every time the values it is dependant on change. For example, if a certain value is very seldomly used, it is a waste of time to keep it up to date for every change. The solution is to let a "Lazy" value have a reference to the value it depends on. It can then read this value and update itself when necessary (when it is used by another value). The contrast is the "Eager" value which is updated and recomputed for every change. Another example of when unnecessary notifications are sent can be seen in figure 4.9, where the first chain of calls are unnecessary.

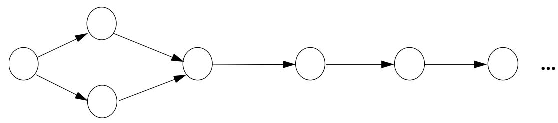

Figure 4.9: Unnecessary updates.

The code below shows how LazyValues are implemented.

```
class LazyValue : public Value {
public:
   LazyValue(float initValue=0) :
      Value (initValue), isValid(false) {};
   operator float() { // update when accessed!
      if (!isValid) {
            myValue = this->Formula();
            isValid = true;
         }
      return myValue;
   }
   void Update(){// Don't Notify if not valid!
      if (isValid) {
            isValid = false;
            this->Notify();
      }
   }
protected:
   virtual float Formula()=0;
   bool isValid;
};
```

Example 4.3: Code for implementing LazyValues.

4.6.3: Asynchronous Communication

Communication with other systems can be handled via interrupts and queues. For example, if a terminal is connected serially to one computer, an interrupt routine can handle the communication from the keyboard and fill a queue. These queues can be treated as values that other values can register on. So far, there has been no need for this solution in the framework.

4.6.4: Priorities

One way to speed up the time-critical values, such as those updating regulators, is to incorporate priorities in the value classes. This means that when a value calls Update for all its dependent values, it does it in priority order. When a value object's Update operation is called, it starts with updating its current value, which could be the actual output port controlling the hardware.

4.6.5: Verifying Real-Time Requirements

In order to verify the real-time requirements in a time-triggered system, there must be a way to put the system in a ”worst-case mode”. The worst case in the Dover framework is when all input values are assigned new values in one single lap of the main loop. In this case all input values notifies all dependants, which in turn notifies all their dependants, and so on. If the system fulfils the time requirements under these conditions, it always will.

4.6.6: Scheduling and Timing

Scheduling in the framework is handled by a clock manager, which also is a value and is dependent on the internal clock (i.e. the real-time clock). Every time the internal clock ticks, the clock manager notifies its dependent values, like timers and clocks, with Tick calls. Each timer and clock knows when its time to notify their dependent values.

Another class in the framework is the PulseValue class. This value acts as a pulse and is set with a Set function call. When the pulse is reset the pulse value send a Notify call to its dependent values. Objects of a PulseValue are used for generating start, stop and clutch in/out signals to secure that the engines and clutches have the necessary time to start and clutch in/out.

The Dover Framework also support a class called InputEventValue, this class reacts on an event and sets its value and notify all its dependent values and then reset itself.

4.7: Value Groups and Control Stands

Individual values are often too small to handle and configure individually, therefore we need some way of grouping them into commonly recurring structures. The design goal is to identify a standard interface which is shared by a number of different concrete structures that implement it differently.

One way to implement this is to have only one access operation with a parameter that indicates the value to access. This flexibility often compensates the drawback with static type checking, which can be used to guarantee that clients only try to access valid values.

Another feature which is needed is to have objects that can act as ”switches” for a set of alternative source values. A switch represents a mapping from a number of inputs to one output. There is also a need for components which map a number of input values to the same number of output values, i.e. CrossBarSwitches. Again the design goal is to find standardised interfaces that both allows and hides different kinds of variations. This ValueGroup Class is used to create control stands. A control stand is a way to configure steering devices, e.g. velocity levers.

4.8: Distribution

The control system built from the framework should be designed as a uniform system, with as little concern as possible put to the physical distribution of the system functions. To provide this possibility the framework should include a networking mechanism that makes the physical distribution of values and objects transparent, and pushes the distribution decision to the last design step. An off-line distribution design is sufficient, i.e. physical split of the system is a part of the design and is not done at run-time.

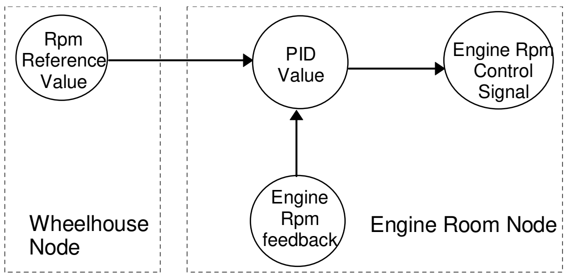

Figure 4.10: The logical design is physically split up on distributed computers.

The design used in the project serves as an interface between the application logic and the network drivers. In this design, a set of values on each node are declared to be globally visible over the network. The Proxy pattern [Gamma95] is used in the design to make it transparent that value objects are located on different nodes.

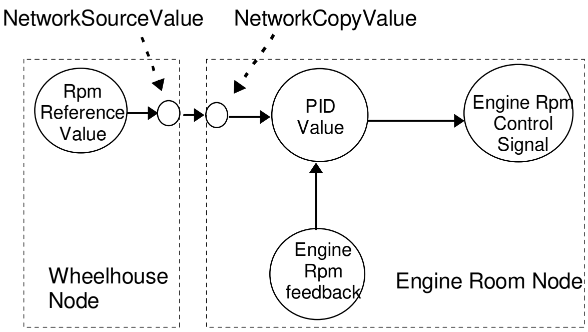

Figure 4.11: NetworkSourceValue and Network-CopyValue are proxies for real values.

When a NetworkSourceValue is changed, its value is propagated to a number of NetworkCopyValues on other nodes. Both kinds of network values co-operate with a NetworkManager object, which encapsulates the particular characteristics of the network used. NetworkCopyValues always exactly replicate the values of their NetworkSourceValues.

On each node there is a NetworkManager object, which knows about the globally visible NetworkSourceValue objects and the NetworkCopy-Value objects on the node. In the simulator application developed, we have used MS-Windows and different programs, communicating via DDE, as different nodes. The NetworkManager subclass needed is therefore DDENetworkManager. The only operation that is network specific is Propagate().

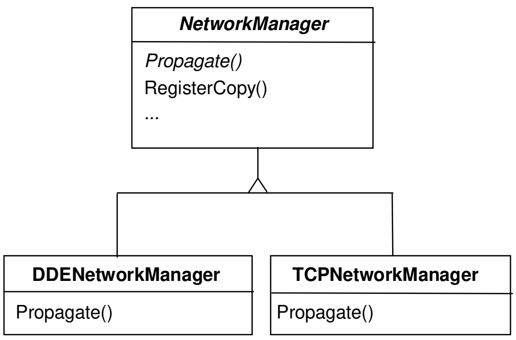

Figure 4.12: The NetworkManager class hierarchy.

4.9: Configurable Parameters and Resources

An important aspect of a vessel manoeuvre control system is the different number of configuration parameters that can be loaded into the program from a text file. The resources are loaded into the system at the start of the program. In this way, the system can easily be changed without rebuilding the program.

The basic idea is that parameters and other resource objects are declared where they are needed and used, as if they were ordinary variables or objects. In the constructor, however, a resource register itself with a central resource manager that can update all resource objects from a file.

The framework can handle two types of resources, constants and transfer functions. A transfer function consists of a number of co-ordinates, and interpolation is used for getting the function value between the co-ordinates. Four types of constants can be handled, integer, float, string and boolean constants.

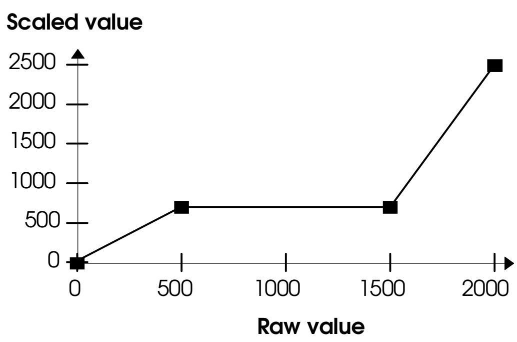

Figure 4.13: Example of a transfer function.

When all the resources of the system has been created, the resources are updated by the ResourceManager. The ResourceManager tells every resource in the file to update themselves. Each type of resource, i.e. constants or transfer functions knows how to update themselves from the file.

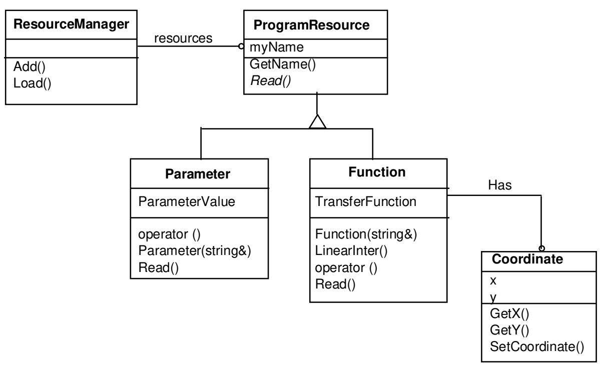

Figure 4.14: The resource managing design.

4.10: Regulators

The regulation of the dynamic items in the target environment like engines RPM and clutch in/out of engines is done with a PID (Proportional, Integration, Derivation) regulator. The class PIDValue, inherits from the abstract class ComputedValue, and has a PID algorithm. The regulation parameters P, I and D are defined in a resource parameter file.

The PIDValue differs from ordinary values that are designed according to the Observer pattern. Instead the PIDValue updates its value with a clock since the regulation algorithm is built on time sample values. The PIDValue generates a limited output control signal to the engine, and the engine RPM feedback is an InputValue to the PIDValue.

4.11: Fault and Alarm Handling

In order to make the value propagation system robust, it is essential that error handling is included in the framework design. Current state of the art typically utilises exception as the basis for error handling. A possible extension to value classes for error handling could be to have dependent alarm values that check if the values are in the correct range. It would also be possible to have redundant values to switch to if the ordinary value is out of range. This is not yet fully implemented in the framework.

4.12: Simulation

The Framework is independent of used platform, and therefore it is possible to simulate the framework on any kind of platform. One way to simulate a vessel control system built from the Dover framework, is to build an interface with Microsoft Windows on a PC and using Windows buttons simulating the engines start/stop, Clutches in/out and control stand choose buttons. The simulation of wheels and levers is done with Windows sliders.

The Dover framework design can also be used for simulation of dynamic environmental behaviour. The input signals from the external environment, e.g. sensors from engines etc., are easily implemented with ”Values”. In this way, the use of external simulators can be avoided, at least in an initial stage. In the Dover project two prototypes have been developed. The second one reused many design details from the first one.

5: Instantiating Applications

The description in this chapter should not be seen as a detailed description but rather as an overview of possibilities when instantiating vessel control systems with the Dover framework. The vessel control system interacts with its environment through a large number of sensors and actuators. This number of sensors and actuators may vary between different systems.

The way to start using the framework is to define a Value instance for each sensor and actuator. Sensors will be instances of InputValues and Actuators will be instances of some subclass of ComputedValue. The InputValues can be initiated with different values. Examples of how to instantiate different kinds of objects are shown in the code fragments below.

```
InputValue        sensorOne(5);
InputValue        sensorTwo(10);
InputEventValue   EventSensor;

Alarm             alarmSensorOne(sensorOne);
IntervalAlarm     intervalAlatmOne(lowLevel,
                     highLevel, sensorTwo);
```

Example 5.1: Instantiating applications.

Nearly all other objects in the program are, in some way, dependent of one or more InputValues, according to the Observer pattern. Objects of the ScaleValue class need to use the parameter file, where the transfer functions are located, to perform the right scaling.

6. Descriptions of Used Design Patterns

Through the later half of the project object-oriented design patterns [Gamma95] has been used. With the introduction of design patterns, the project found the right level of abstraction and the possibility to design a good framework. The most valuable patterns have been Observer (which we use for Values), Singleton, Proxy and State. In the sections below, more detailed information is given on the design patterns we have used, and in some cases how we have adapted them to suit our problems.

6.1: The Observer Pattern

A common side-effect of partitioning a system into a collection of co-operating classes is the need to maintain consistency between related objects. You don‘t want to achieve consistency by making the classes tightly coupled, because that reduces their reusability. The solution is to use the Observer pattern [Gamma95] to define a one-to-many dependency between objects, so that when one object changes state, all its dependants are notified and updated automatically

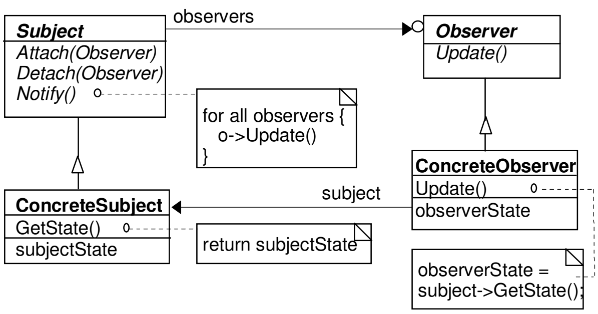

Figure 6.1: The Observer pattern structure.

Subject:
- knows its observers. Any number of Observer objects may observe a subject.
- provides an interface for attaching and detaching Observer objects.

Observer:
- defines an updating interface for objects that should be notified of changes in a subject.

ConcreteSubject:
- stores state of interest to ConcreteObserver objects.
- sends a notification to its observers when its state changes.

ConcreteObserver:
- maintains a reference to a ConcreteSubject object.
- stores state that should stay consistent with the subject‘s.
- implements the Observer updating interface to keep its state consistent with the subject‘s.

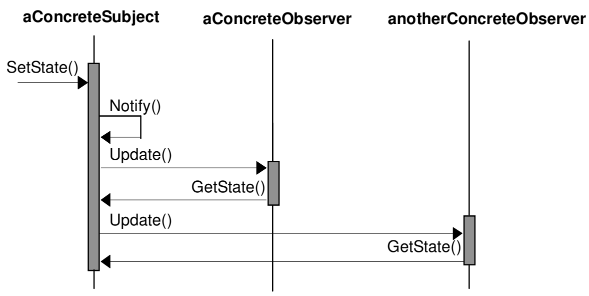

Figure 6.2: Interaction diagram of the Observer pattern.

First the SetState() operation in the aConcreteSubject is called. The aConcreteSubject object changes state and the Notify() operation is invoked, calls the Update() operations on all its Observers. The Observers call the GetState() operation in the subject and get the subjects state in return and make a re-computation of itself.

6.2: The Singleton Pattern

A design pattern used in several places in the Dover framework is the Singleton pattern [Gamma95]. The intent is to ensure that a class only has one instance, and provide a global point of access to it. For some classes it is important that they have exactly one instance. Although there can be many NetworkValueSources in a system, there should only be one NetworkValueManager. One possible solution is to have a global variable that makes the instance available, but that does not keep you from instantiating multiple objects.

A better solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created, and it can provide a way to access the instance.

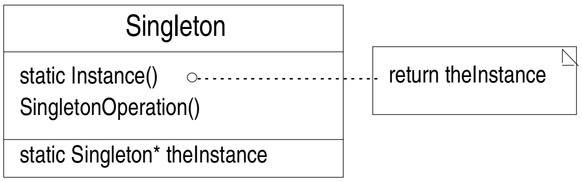

Figure 6.3: The Singleton pattern structure.

```
class Singleton   {
public:
   void example(void) {cout << "Singleton";}
   static Singleton* Instance() {
      // the first call creates the instance
      if (_instance == 0) {
         _instance = new Singleton;
      }
      return _instance;
   }
protected:
   // can not create object from outside
   Singleton() {}
private:
   static Singleton* _instance;
};
Singleton* Singleton::_instance = 0;

// Access by using Instance()
   Singleton::Instance()->example();
```

Example 6.1: Implementing the Singleton pattern.

6.2.1: Subclassing Singleton

In the NetworkManager design, we had a need to subclass into DDENetworkManager. The problem was that the constructor in NetworkManager did not know that it should create a DDENetwork-Manager. This forced us to modify the design and add an operation to the DDENetworkManager class which creates a DDENetworkManager object.

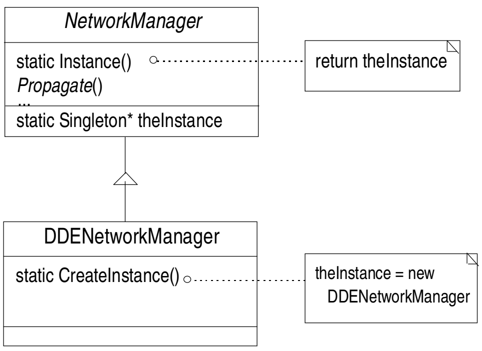

Figure 6.4: The Singleton pattern used in the NetworkManager class.

6.3: The Proxy Pattern

The Proxy pattern [Gamma95] is used in the design to make it transparent that value objects are located on different nodes. Proxy is applicable whenever there is a need for more sophisticated reference to an object than a simple pointer.

- *Problem*: Sometimes you want to have a pointer to something that you can‘t refer to directly, e.g. objects on other machines or on file.
- *Solution*: Provide a proxy (or surrogate) that controls access to the real object.

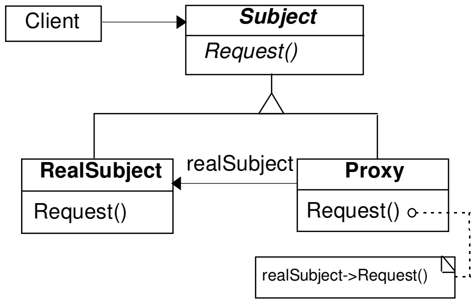

Figure 6.5: The Proxy pattern structure.

The Proxy class and the RealSubject class have the same interface defined in the Subject class. The client code access a proxy object, which is responsible for the final access.

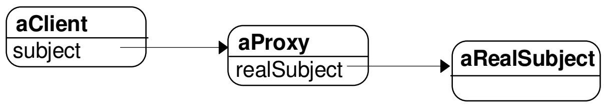

Figure 6.6: Instance diagram for the Proxy pattern.

6.4: The State Pattern

The use of state machines is an effective way to implement changes in a object's behaviour at run-time depending on its current state. The design of the state machines, in the Dover framework, is done according to the State pattern [Gamma95].

- *Problem:* Sometimes we want to change an objects behaviour when its internal state changes (as if it changed its class)
- *Solution:* Polymorphism lets a message be implemented by different methods in different classes. Have one subclass for each state !

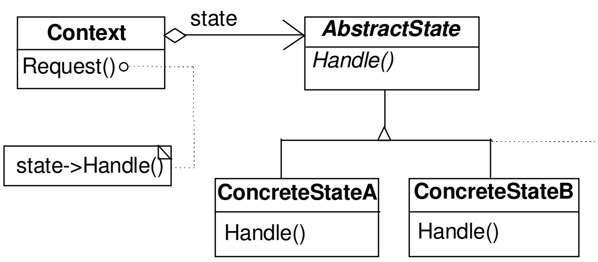

Figure 6.7: The State pattern structure.

In the Dover framework, engines are controlled with state machines. An engine can, for instance, have Off, Starting, Running and Stopping states.

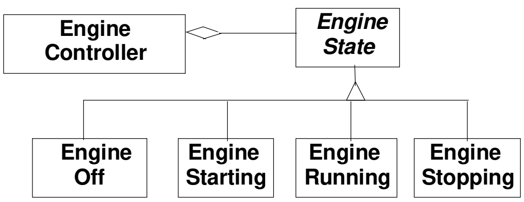

Figure 6.8: The State pattern as used in the Dover framework.

The Controller object has a pointer to its current state. The change of state is done with the ChangeState operation.

7: Outcome of the Project

The personnel at Karlskronavarvet AB have made the transition from structural design and C programming to skilled object-oriented designers and C++ developers. The transition has not been easy, but has been done. All project members have been educated in object-orientation, framework and pattern technology. Karlskronavarvet has a framework for vessel control systems and the intention is to extend and refine it in the future.

The framework has, so far, only been used on the PC platform, which is a drawback since the systems have other target platforms (Motorola processors, VME-bus). The intention was to let persons not involved in the framework development process, develop small applications with help of the documentation. Unfortunately this has not been done in the scope of the project.

With the diagram in figure 7.1, we want to show the benefits we estimate are possible to obtain with use of the Dover framework. The reason for these figures being just estimates is that the project has only developed two prototypes, i.e. no real target platform system. But all the framework code is fully portable, which makes the estimations based on prototype development realistic. As part of the project, we will write a follow-up report in two years time after the development part ended. Then we will have more detailed metrics to present.

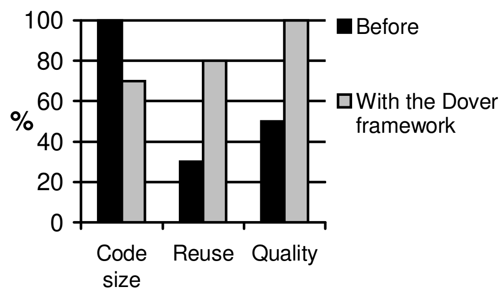

Figure 7.1: Estimated figures on project result.

The "Code size" is the total amount of code in the system. The code size is lower with the framework because of object-oriented features such as data abstraction, inheritance and dynamic binding. Our experience is that most of the coding effort is configuration of the system, when the framework is used.

The "Reuse" column is the percentage of the total system that can be reused from previous systems. We estimate that about 80% of the system can be reused, i.e. 80% of the code is in the framework or reusable parts outside.

The last column shows how the "Quality" in a system will increase with the use of the framework, mostly because less new code needs to be developed. With quality we mean number of bugs or errors in the development of a system. We believe that the quality improvement is the most important aspect when investing in framework technology.

7.1: Documentation

Three documents has been produced :

- Technical Overview - give insights to the domain and the framework.
- Development Handbook - to be used by the application developers.
- Experience Report

The Technical Report has the following contents:

Introduction
  Scope, target audience, required knowledge
Domain Description
  General Aspects
  Variant Aspects
Design Principles
Description of the Framework
Instantiating Applications
Implementation Details

The Development Handbook has the following contents:

Instantiating an application from existing components

Recipes for adapting to different variant aspects (based on the variant aspects described in Technical Overview)

7.2: Experiences

- Object-orientation and C++ programming were more difficult than we thought.
- Most available textbooks were insufficient, they mostly focus on notation and a somewhat ”naive” process. The established methods build on the traditional one-way flow of control. They support inheritance between classes but are weak on dynamic binding. We found that the notation can be used to describe the design but the method itself can do more harm than good. The break-through for the Dover project came with the Design Patterns book [Gamma95].
- When the pattern theories were introduced we all were excited but it was hard to comprehend in the beginning and to use it in our design. You have to try them out yourself, just reading the book is not enough.
- The iterative and incremental planning strategy was successful. We started with a waterfall approach, but soon realised that it is unsuitable for a project which aims at research and building knowledge.
- External expertise was very important and resulted in many new solutions to the design problems we had.
- There is a need for good knowledge in the domain and in object-oriented technology to succeed in framework development. The key point in designing a framework is to identify a good level of abstraction and to find the right variant aspects. We chose ”Value” as the base level.
- Research projects with no short term profit have lower priority and project members are pulled in and out of the project. You need to have management support!

If you work in a specific domain and develop applications that are slightly differing, you could benefit from investing in framework development. Below, we have listed some steps that we think you need to go through.

- Write a program with an existing framework - you can not start to develop a framework if you haven’t used one.
- Study design patterns to become a better designer.
- Write a simple framework of your own and test it with small programs.
- Develop a small domain-specific framework.
- Use your framework in more than one project.
- Try to get other developers to use it.

8: Conclusions

Developing frameworks allows you to solve a problem once and then reuse that solution any time you face a similar problem. After you have captured domain expertise in a framework, you and others can reuse that framework, saving time and money both in program development and in maintenance. Thanks to dynamic binding, calls can go from the framework code to the user code, which makes it possible to have reuse on a substantially higher level, compared to using class libraries. We believe that the most important factor is quality improvement. When you have a mature framework, only a fraction of the total code for a new system needs to be added, which naturally limits the possibilities of introducing flaws. With the introduction of Design Patterns in software engineering, and particularly in object-oriented design, we think that a major step has been taken forward in producing reusable and high quality software.

In the Dover project a vessel control system framework has been developed, with successful results. We believe a lot in this technology and expect that we will use it in other domains as well.

Acknowledgements

We would like to thank Mr. Lennart Ohlsson for his contributions as an expert in object-oriented design and framework technology. We have had great help from the books [Gamma95], [Mattsson96], [Pree95], [Cotter95], and from the patterns home page:

http://st-www.cs.uiuc.edu/users/patterns/patterns.htm

This project was financed by the European Commission, DG III, ESPRIT III/ESSI.

References

[Alex79] "The Timeless Way of Building", C. Alexander, Oxford University Press, New York, 1979.

[Boehm88] "A Spiral Model for Software Development and Enhancement", B. Boehm, IEEE Computer, vol. 21, no. 5, May 1988, pp. 61-72

[Booch91] ”Object Oriented Analysis and Design with Applications”, G. Booch, Benjamin/Cummings, 1994, Second Edition.

[Coad92] "Object Oriented Patterns", P. Coad, Communication of the ACM, September 1992.

[Cotter95] ”Inside Taligent Technology”, S. Cotter et. al., Addison Wesley, 1995.

[Eriksson94] ”An Object-Oriented Framework for the Design of Hard Real-Time Systems - A study focused in RealTimeTalk”, C. Eriksson, Royal Institute of Technology, KTH, Stockholm, Sweden, 1994

[Gamma95] "Design Patterns - Elements of Reusable Object-Oriented Software", E. Gamma, R. Helm, R. Johnson, J. Vlissades, Addison-Wesley, 1995, ISBN: 0-201-63361-2

[Jacobson92] ”Object-oriented Software Engineering, A Use Case Driven Approach”, I. Jacobson et. al., Addison Wesley, 1992.

[Johnson92] "Documenting Frameworks using Patterns", R. Johnson, OOPSLA '92 Proceedings, Vancouver, October 1992.

[Kopetz92] ”Sparse Time versus Dense Time in Distributed Real-time Systems”, H. Kopetz. The 12th International Conference on Distributed Computing Systems, Japan, June 1992.

[Mattsson96] ”Object-Oriented Frameworks - A survey of methodological issues”, M. Mattsson, University College of Karlskrona-/Ronneby, WWW home page: http:/ /www.pt.hk-r.se/~michaelm

[Pree95] ”Design Patterns for Object-Oriented Software Development”, W. Pree, Addison-Wesley, 1995

[Rumb91] "Object-Oriented Modeling and Design", Rumbaugh et. al., Prentice Hall, 1991, ISBN: 0-13-630054-5

