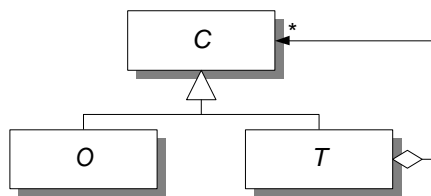


*Designing Event-Controlled
Continuous Processing Systems*
COT/2-41-V1.0



Centre for Object Technology

*Centre for
Object Technology*

Revision history: 99.09.06 V.0.9 Paper to be presented on the
 Embedded System Conference Europe
 Nov. 1999 in Maastricht
 99.12.22 V1.0 Final version for publication

Authors: Hans Peter Jepsen, Danfoss Drives
 Finn Overgaard Hansen, Teknologisk Institut

Status: Final

Publication: Public publication

Summary:

This paper will present the results of an object-oriented analysis, design and partial implementation of a frequency converter.

The paper will also present an outline of an architecture for this kind of system and describe how the combination of several Design Patterns (Gamma et. al.) have been useful for building an object-oriented architectural model described with UML notation.

The results presented will be relevant for most event-controlled continuous processing systems.

The paper was presented at the Embedded System Conference Europe, November 16-18, 1999 in Maastricht.

© Copyright 1999

Designing Event-Controlled Continuous Processing Systems

Hans Peter Jepsen, Danfoss Drives,
(E-mail: hans_peter_jepsen@Danfoss.com)
Finn Overgaard Hansen, Danish Technological Institute,
(E-mail: finn.overgaard.hansen@teknologisk.dk)

ABSTRACT

Many embedded systems, such as systems in the process control and measurement domains, are designed to perform continuous processing of input values and to produce and maintain corresponding output values. At the same time these systems have to react on events that reconfigures the processing algorithms.

This paper will present some of the results of a Danish research project entitled “Architecture of embedded systems” inside COT (Centre for Object Technology). More specifically the paper will present the results of an object-oriented analysis, design and partial implementation of a frequency converter. The paper will also present an outline of an architecture for this kind of system and describe how the combination of several Design Patterns (Gamma et. al.) have been useful for building an object-oriented architectural model described with UML notation. The results presented will be relevant for most event-controlled continuous processing systems.

1. INTRODUCTION

This paper presents some architectural concepts that we believe are relevant for most event-controlled continuous processing systems. They are the results of a pilot project on the OO-development of a frequency converter in the Danish research project COT. Allow us to elaborate on the frequency converters and on COT, just to give you some valuable background information.

A *frequency converter* is a device used to control a normal asynchronous electro motor, so that the motor speed or motor power is exactly, what is needed in a given situation. A frequency converter is often called a “drive”. Two typical applications will illustrate the use of a drive. The first example, shown on Figure 1, is the use of a drive to control a conveyer belt.

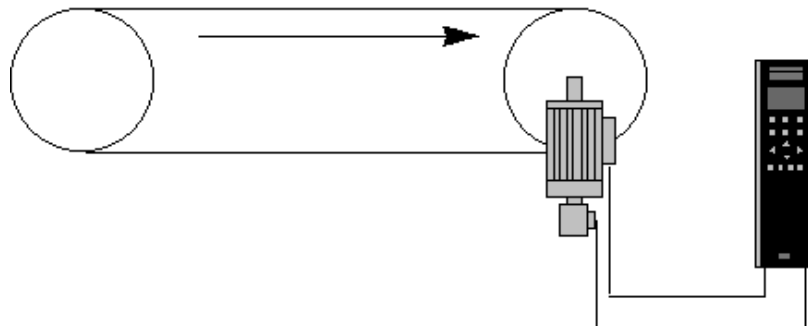


Figure 1. A drive used for controlling the speed of a conveyer belt.

The purpose of using a drive in connection with a conveyer belt is normally to ensure that the speed of the belt is exactly what is needed, but also to keep a constant speed regardless of the load on the belt. When changing the speed (e.g. start and stop), it is crucial that the changes are “smooth”.

Advanced frequency converters can be used in a large number of applications and have a high degree of complexity. As an example the software for the VLT® 5000 series of frequency converter consists of approx. 150.000 lines of C-code. The hardware design is based on an ASIC, which - measured by the number of gates - has a complexity as an 80386 CPU. (VLT® is the trademark for drives produced by Danfoss Drives).

The *Centre for Object Technology (COT)* is a Danish collaboration project with participants from industry, universities and technological institutes. The overall objective of Centre for Object Technology (COT) is to conduct research, development and technology transfer in the area of object-oriented software construction, for more details see [COT]. The activities in COT are based on the actual requirements of the participating industrial partners.

COT is organised with 6 industrial cases, one of them is case 2 entitled “Architecture of embedded systems”. Producers of embedded systems are confronted with increasing demands towards complexity, quality and fast and predictable development time. In order to meet some of these requirements, many producers are interested in switching to object-technology in order to obtain some of the advantages of OO technology. The purpose of COT case 2 is therefore to experiment with the introduction of object technology in embedded systems.

A major activity inside COT case 2 is a number of pilot collaboration projects among industrial partners (Bang & Olufsen, Danfoss Drives and Danfoss Instruments), the Danish Technological Institute and the Department of Computer Science at Aarhus University. The purpose of these projects is to experiment with object-oriented software development within the domain of an industrial partner. In the pilot project at Danfoss Drives, an object-oriented model and corresponding prototype implementation in C++ for the central parts of a VLT frequency converter were developed.

2. OVERVIEW OF THE TWO-PART ARCHITECTURAL MODEL

A major breakthrough in the modelling of our frequency converter was the formulation of a conceptual model, which we call “the two-part architectural model”. The following figure will illustrate the ideas.

The part *below* the dashed line represents the part of the system that is responsible for the continuous data processing, e.g. process control and measurement. In a given situation (determined by certain setting of the “switches”) the data is flowing through a subset of the “boxes”. Some of the characteristics of this part are 1) that it has data flow architecture and 2) it is normally driven by periodic interrupts.

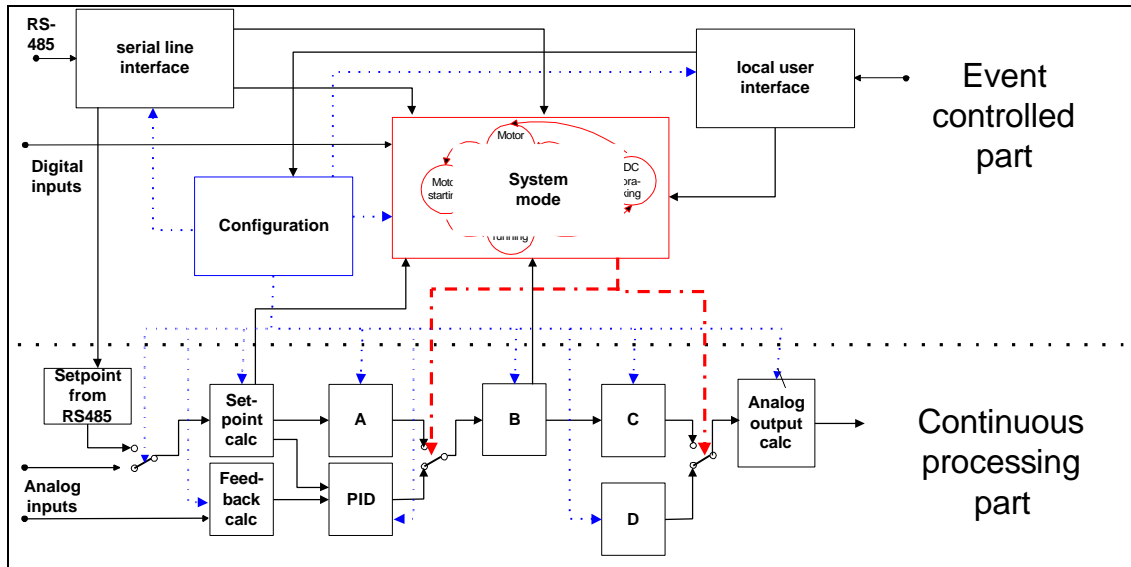


Figure 2. The two-part architectural model

The part *above* the dashed line represents the part of the system responsible for handling events occurring asynchronously with the continuous data processing. State machines normally dominate this part, and configuration data will typically also be maintained in this part.

The event-controlled part configures (parameterises) the “boxes” in the continuous data processing part (represented with the arrows to the boxes), but also determines the signal path in this part (represented with the arrows to the switches). On the other hand the continuous data processing part can produce events that have to be handled in the event-controlled part (represented with the up-arrows).

To illustrate the two-part architectural model we will show, how elements from a frequency converter can be mapped onto this model.

The mapping of drive-software required for the conveyer belt application onto the two-part architecture is as follows: When the conveyer belt is running, the motor speed is controlled by the continuous data processing part. Configuration values for the “boxes” are delivered to these boxes from the Configuration subsystem of the event-controlled part. When a stop command is delivered to the drive, this command arrives in the event-controlled part where it is determined that the algorithm in the continuous processing part has to be replaced. As a result the setting of some “switches” in the continuous part are changed.

The second application example, shown on Figure 3, is the use of a drive to control the speed of a ventilator.

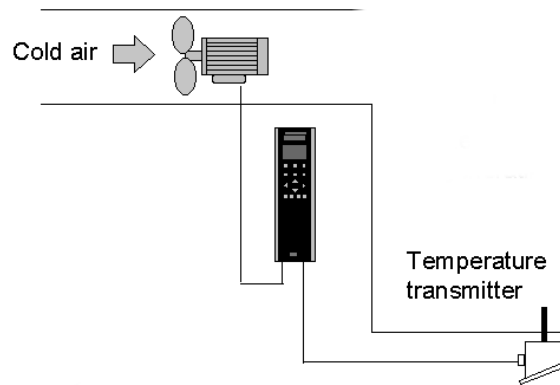


Figure 3. A drive used for controlling the speed of a ventilator.

The purpose of using a drive in connection with a ventilator is normally to maintain a desired room temperature. It is possible to save a lot of energy by reducing the ventilator speed as much as possible while maintaining the desired temperature. In this application the ventilator is “windmilling” when the motor is not powered. Therefore, before applying power, the ventilator must be stopped or “caught”.

The mapping of drive-software required for this application onto the two-part architecture is as follows: Initially the motor is coasted and the ventilator is windmilling. When the event-controlled part of the drive receives a start command, it sets the switches in the continuous part to perform an algorithm, that we call “catch spinning motor”. Whenever this algorithm has detected the motor speed and adjusted its output to the motor, we have reached a situation where the drive is controlling the motor. The algorithm sends an event to the event-controlled part with this information. The event-controlled part then changes the settings of the continuous part to perform a “process closed loop” algorithm.

3. THE EVENT-CONTROLLED PART OF A VLT APPLICATION

Analysing and designing the event-controlled part of the SW is well documented in existing literature and need no further explanation.

It is a well-known fact that the use of state machines is a very important design technique for this part. Our experience shows that compared to the modelling of state machines in structured analysis and design, an object-oriented method seems to result in a larger number of state machines with each state machine being smaller and more understandable. There are two reasons. Firstly, the UML state charts are more expressive than SA/SD state charts. Secondly, the emphasis on the responsibility of each class tends to move the state machines from the system level to the class level.

4. THE CONTINUOUS DATA PROCESSING PART OF A VLT APPLICATION

We have found, that the architectural styles “Process Control” and “Pipes and Filters” [Shaw&Garlan96] have helped to understand and describe the continuous data processing part of our system. We will illustrate it with the two applications described above.

We will first examine the ventilator-control application presented on Figure 3. The Process Control model from [Shaw&Garlan96] - in this case a closed loop control – is shown on Figure 4 with the relevant terms for this application.

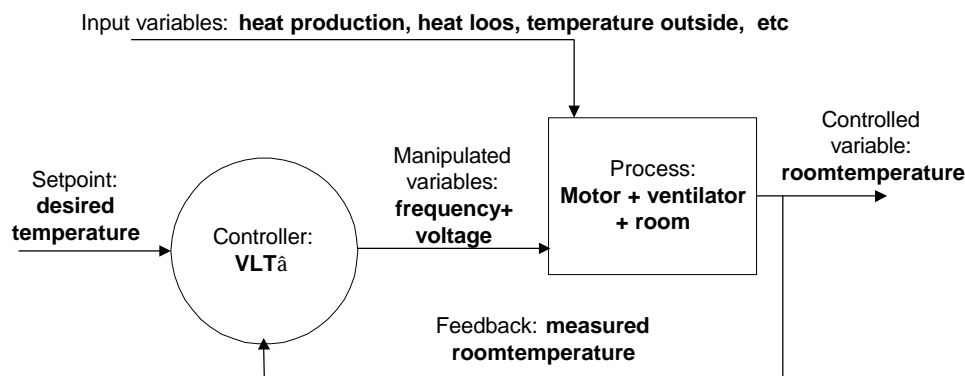


Figure 4. Closed loop feedback process control of room-temperature

The responsibility of the software required for this application is shown on Figure 5 as a block diagram where the “pipes and filters” structure is easily recognised.

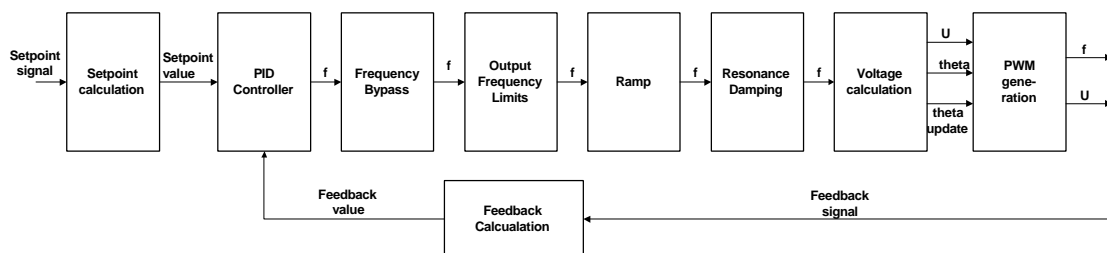


Figure 5. Block diagram of a VLT’s “Process closed loop” algorithm

A few comments to some of the processing blocks on Figure 5. The four blocks “Frequency Bypass” to “Resonance Damping” does possibly change the desired output frequency, based on configuration values given by the user (e.g. “Ramp” will keep the changes in the output frequency below a value, set by the user). PWM generation (implemented in the ASIC) is a “continuous” production of a voltage vector, which has the desired amplitude and frequency.

We get an almost similar model by looking at the conveyer belt application (Figure 6) where the speed of the belt is controlled without speed feedback (the speed is estimated by measuring the motor's current consumption). In this case the process control is an example of a closed loop feed-forward speed control and the algorithm is called "Speed open loop".

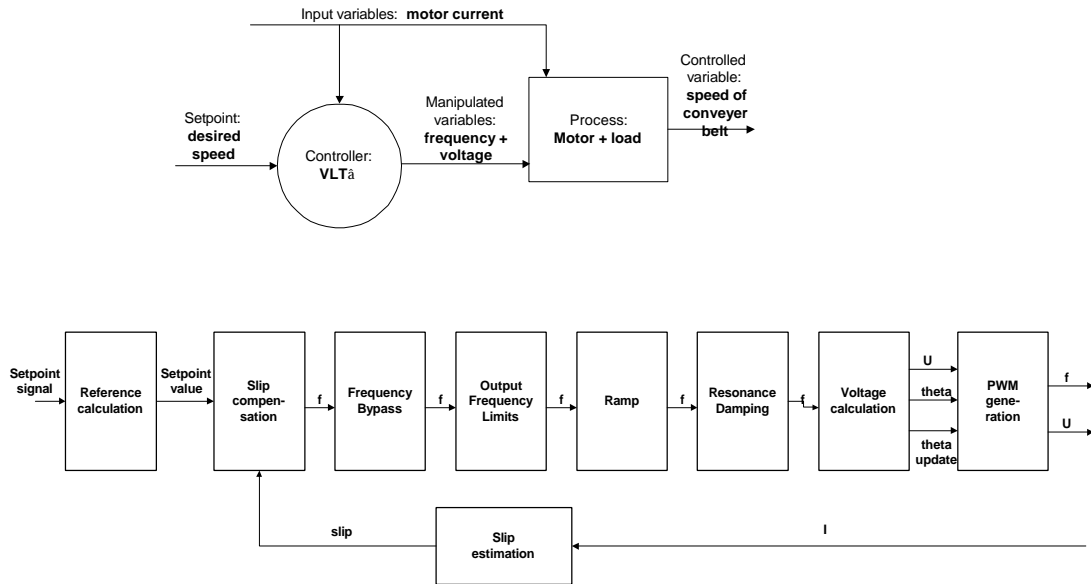


Figure 6. Block diagram of a VLT's feed-forward speed control – "Speed open loop"

We are now ready to examine the two-part architecture introduced above. If our frequency converter should only handle the two types of applications described, the continuous data processing part of the software would contain the elements below the dashed line on the following figure.

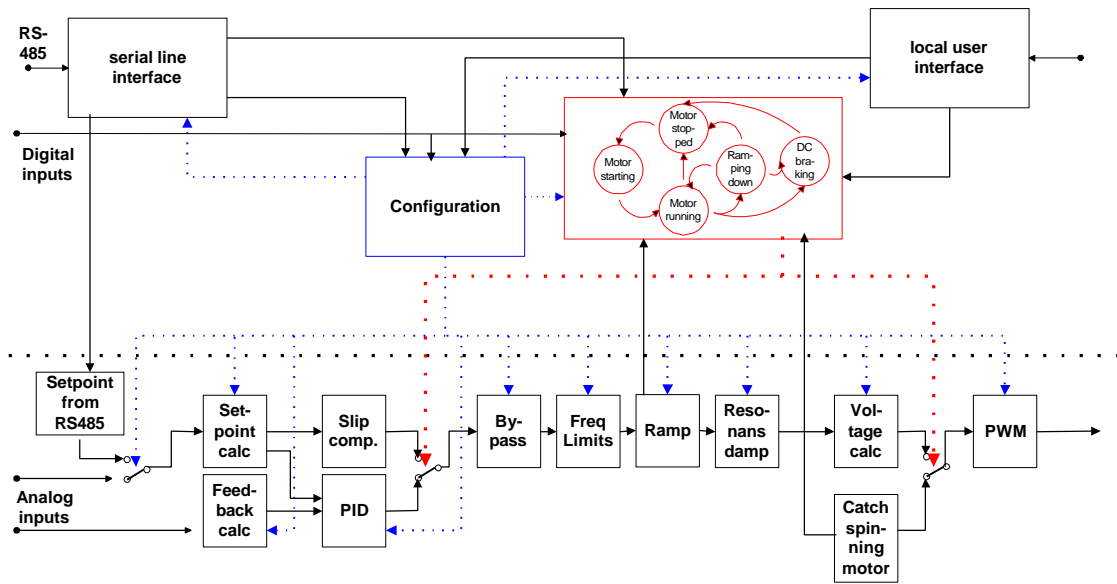


Figure 7. Architecture view of drive software - very simplified

5. OBJECT-ORIENTED DESIGN OF THE TWO-PART ARCHITECTURAL MODEL

After introduction of the pilot project domain and a high level description of the two-part model, the following sections will describe how this model was realised with an object-oriented design with heavy use of Design Patterns [Gamma95] and documented using the UML notation [UML97].

Figure 8 shows a UML package diagram of the two-part model for the informal architecture diagram presented on Figure 7. The event-controlled part is further divided into a VLT control package and a configuration package both affected by discrete events or commands from the VLT user. The continuous processing part is divided in two different continuous processing packages; one for controlling the Motor and the other for supervising purposes, both dependent on a configuration package.

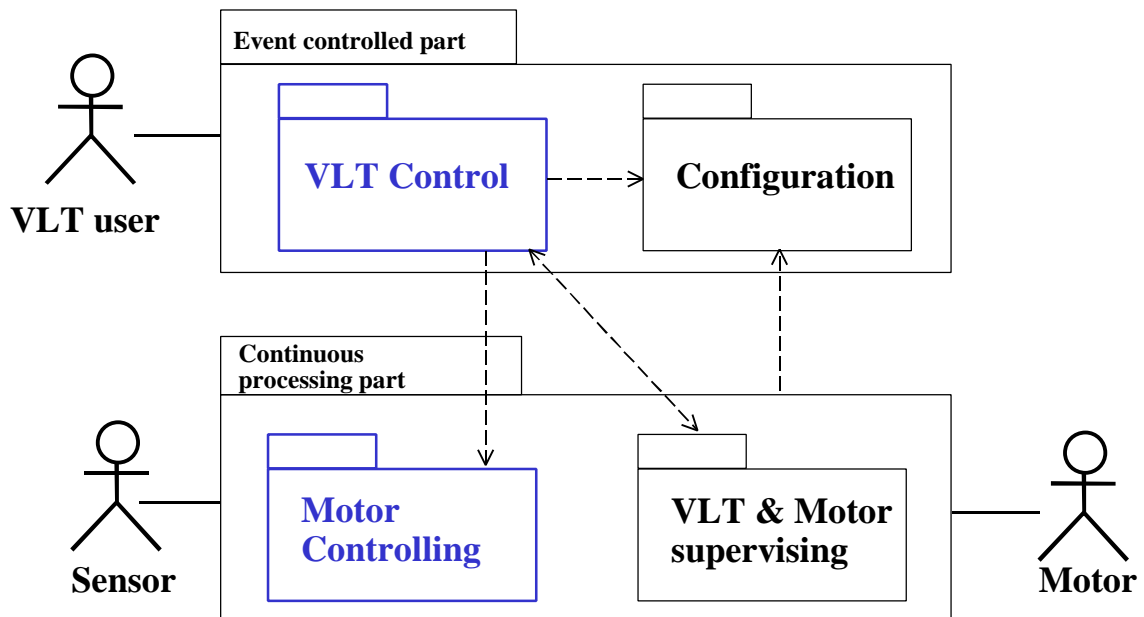


Figure 8. UML package diagram showing the two-part model

5.1 The continuous processing part

The continuous processing part is designed as a combination of the Strategy design pattern [Gamma95] combined with the Pipes and Filter design pattern [Shaw & Garlan96] and [Buschmann96]. Figure 9 shows the general structure for the Strategy pattern according to [Gamma95]. In this pattern the Context object delegates a part of its algorithm to its connected strategy object. This solution makes it possible for the context object to change its algorithm by changing the concrete strategy object pointed to by the context object. The actual implementation of the Strategy in the pilot project is shown on Figure 10. The different strategies in this implementation correspond to the different VLT operation modes i.e. Speed open loop processing, Process closed loop processing etc. The *AlgorithmInterface* function in the Strategy Class is here called *generateOutput*.

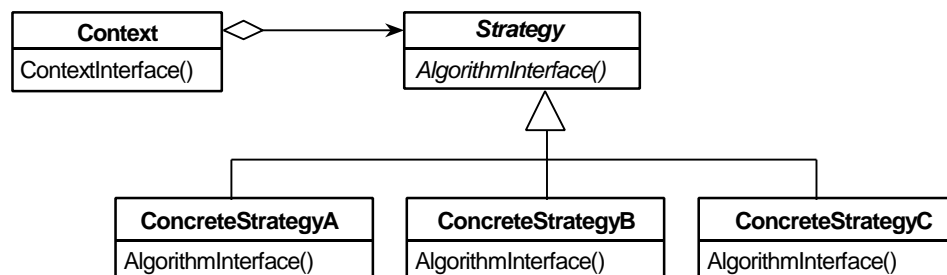


Figure 9. Structure for Strategy pattern

The function *generateMotorOutput* in the Class *MotorOutputGenerator* will be activated continuously by a timer interrupt every 2 ms. This function will delegate the generation of motor output data to the current active output controller objects *generateOutput* operation.

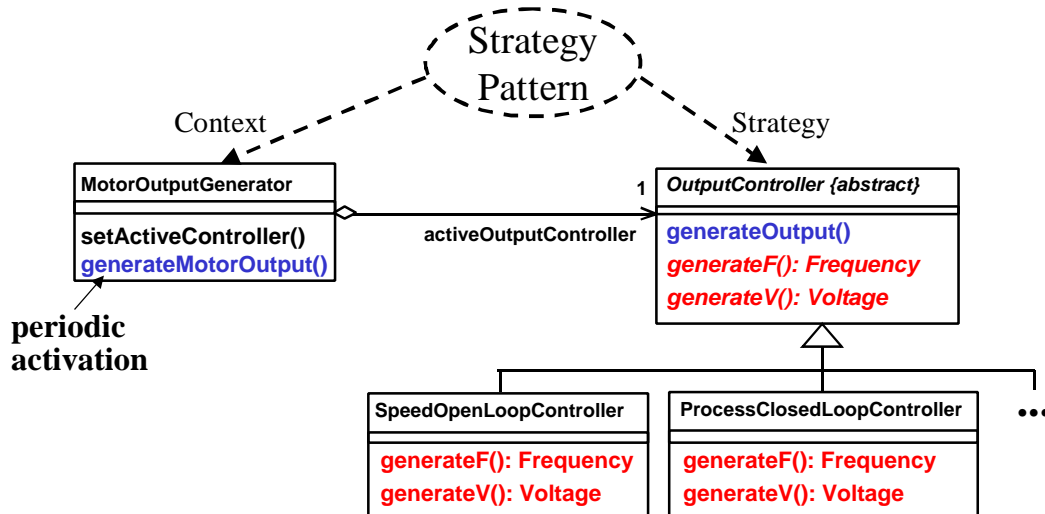


Figure 10. Class diagram showing realisation of Strategy Pattern

```

MotorOutputGenerator::generateMotorOutput()
{
    theActiveOutputController->generateOutput();
}
  
```

The *generateOutput* operation calculates the control parameters and sends them to the ASIC controlling the motor.

```

OutputController::generateOutput()
{
    frequency= generateF(); // pure virtual function
    voltage= generateV(); // pure virtual function
    thePwmAsic->output(frequency,voltage);
}
  
```

In this implementation, the strategy operation *generateOutput* is also a Template Method [Gamma95] where the two functions *generateF* and *generateV* are pure virtual (C++) functions which must be implemented with code in subclasses of the *OutputController* class.

The VLT system requires as smooth a shift as possible from one type of controller to another – called bumpless transfer. The solution to this requirement was to enhance the standard Strategy pattern with functionality to enable this bumpless transfer between the operation modes. This is implemented with the addition of two operations *activate* and *deactivate* in the *OutputController* class.

The configuration of the “strategy” is performed by calling the operation *setActiveController* in the class *MotorOutputGenerator*, with the new *OutputController* object as parameter. The current controller is deactivated and the returned information is used to activate the new controller object, which starts with the same conditions as the previous controller.

```
MotorOutputGenerator::setActiveController
                        (OutputController: newController)
{
    controllerInfo= theActiveOutputController->deactivate();
    theActiveOutputController= newController;
    theActiveOutputController->activate(controllerInfo);
}
```

The different VLT operation modes require identical processing components for the controlling parts of the overall algorithm. Using the Pipes and Filter architecture pattern described in [Shaw&Garlan96] and [Buschmann96] fulfils this requirement. It results in a flexible design where the algorithm components can be easily configured to the actual needs of a particular operation mode.

The Pipes and Filter architecture pattern divides an algorithm into several sequential processing steps, which are connected by data flow, the output of one step is input to the subsequent step. Each processing step is implemented by a *filter* component. Each filter component is connected to another filter component by a *pipe* mechanism. Each filter processes its input data and forwards it to the next filter via the pipe.

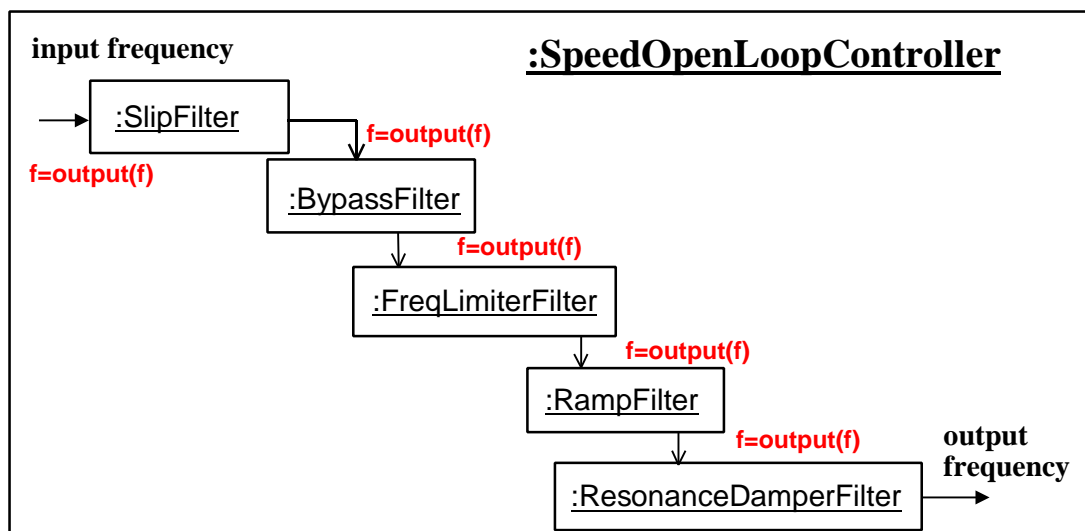


Figure 11. Object diagram for Speed Open Loop operation mode controller

Figure 11 shows an example of a UML object diagram for a Speed Open Loop Controller with the different filter components modelled as objects (compare with block diagram on Figure 6). This is an example where a UML object diagram is very useful to

illustrate the runtime situation where the corresponding UML class diagram only show the static code organisation.

The corresponding object diagram for a Process Closed Loop controller (presented in Figure 5) will be exactly the same except that the first object will be replaced by a PidFilter object which encapsulating a PID algorithm.

Figure 12 shows the static structure of the corresponding class diagram implementing the Pipes & Filter pattern.

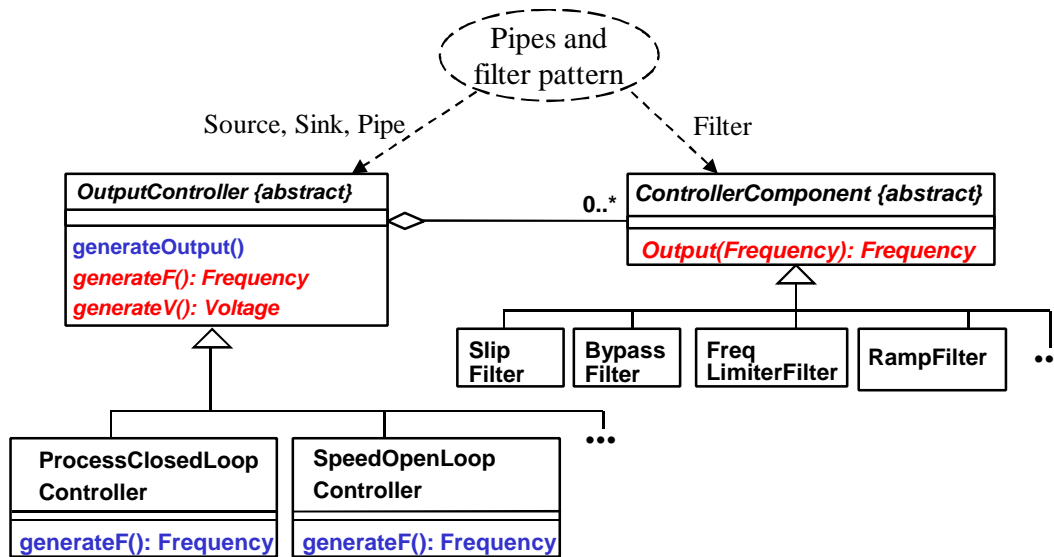


Figure 12. Class diagram showing realisation of Pipes and Filter pattern

The class *OutputController* (the strategy class) is an aggregation of different Filter classes from The Pipes & Filter pattern. Each of these filter classes implement parts of the final algorithm and during runtime it is possible to combine them with other filter objects to obtain the desired functionality for a given controller.

The actual implementation of the pipes and filter patterns is very simple as the pipes are implemented in the code by a certain call sequence of the filter functions. The following C++ code shows the implementation for the *generateF* operation in the *SpeedOpenLoopController* class shown on Figure 12.

```

Frequency SpeedOpenLoopController::generateF()
{
    frequency = theSlipFilter->output(frequency);
    frequency = theBypassFilter->output(frequency);
    frequency = theFreqLimiterFilter->output(frequency);
    frequency = theRampFilter->output(frequency);
    frequency = theResDamperFilter->output(frequency);
    return frequency;
}
  
```

Figure 13 shows the collaboration of the different objects involved in the continuous processing loop, which is activated every 2 ms by a call of the *generateMotorOutput* operation. The use of polymorph operations results in fast processing as the execution path through the code is set up in advance by the event controlled part (by call of *setActiveController*). The traditional non-OO approach in the C programming language is to have a substantial number of C switch statements in the code, where each switch is calculated in the continuous processing loop.

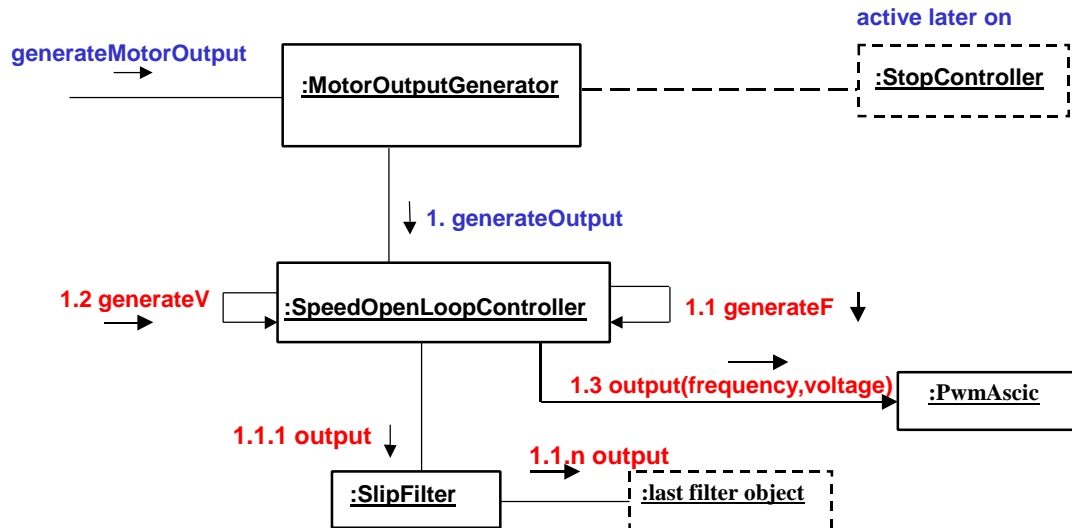


Figure 13. Object collaboration diagram for the continuous processing

5.2 The event-controlled discrete part

This part is activated by external and internal events, which may occur at any discrete point in time. Examples of user-initiated events are a start command to the system, a change of the current operation mode or a change of configuration parameters. Another source of events is the different monitoring functions supervising the VLT and the connected motor and generating internal events when some predefined situations occur.

Figure 14 shows a *MotorManager* class, with the responsibility of controlling the motor by configuring the continuous part with the right controller. The *MotorManager* class is designed as a state machine modelled with a UML State Chart.

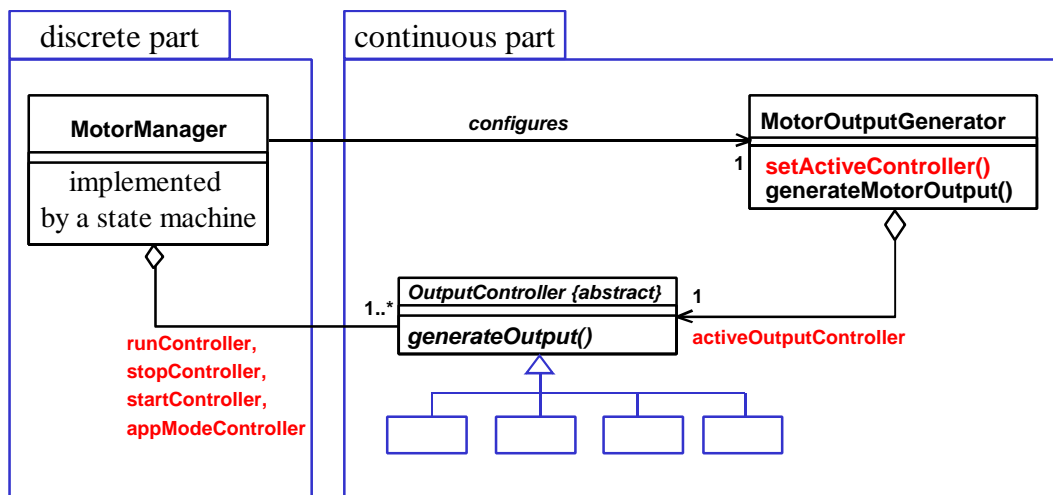


Figure 14. Class diagram with the configuration of MotorOutputGenerator

Figure 15 shows a part of the State Chart for the *MotorManager* class. The configuration of the continuous control part is performed either on a state transition or when a new state is entered. The object pointers *runController* and *stopController* are initiated to point at the actual run- and stop controller objects in another part of the program depending on the actual configuration parameters.

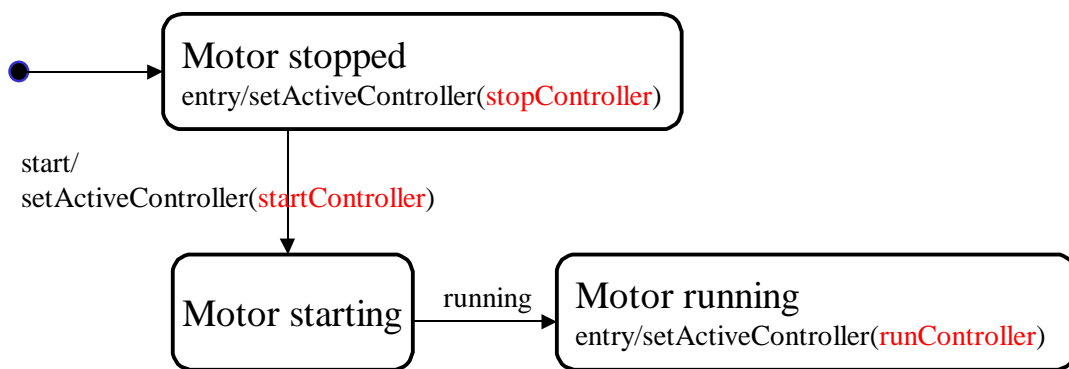


Figure 15. Part of the State Machine for the MotorManager class

The actual state machine implementation is realised by using the State design pattern [Gamma95] working in concert with the Command design pattern [Gamma95]. In the State design pattern each state of the actual state machine is modelled as a class with the triggering events as operations.

Each external event is transferred to a command object, which is subsequently sent to the MotorManger class for processing. This solution resulted in a simple interface for the MotorManger class obtained by the operation `handleCommand(Command* theActualCommand)`.

6. Method considerations

This section will describe how the two-part model approach can be used as a constructive step in the development process, starting with requirement specification using the Use Case technique [Jacobson92].

6.1 Use Cases

The Use Case technique was employed in the pilot project to specify the VLT functionality. Use Cases for this system type can be divided in two different groups; one group for the event controlled part of the system functionality and the other group representing the continuous processing parts.

The Use Case diagram on Figure 16 shows two examples of event initiated Use Cases, where the Use cases are activated by events from external actors to the system i.e. the VLT user. The lower part of the figure shows two examples of the second group of Use Cases, corresponding to the continuous processing parts, where the Use Cases are continuously activated by the system.

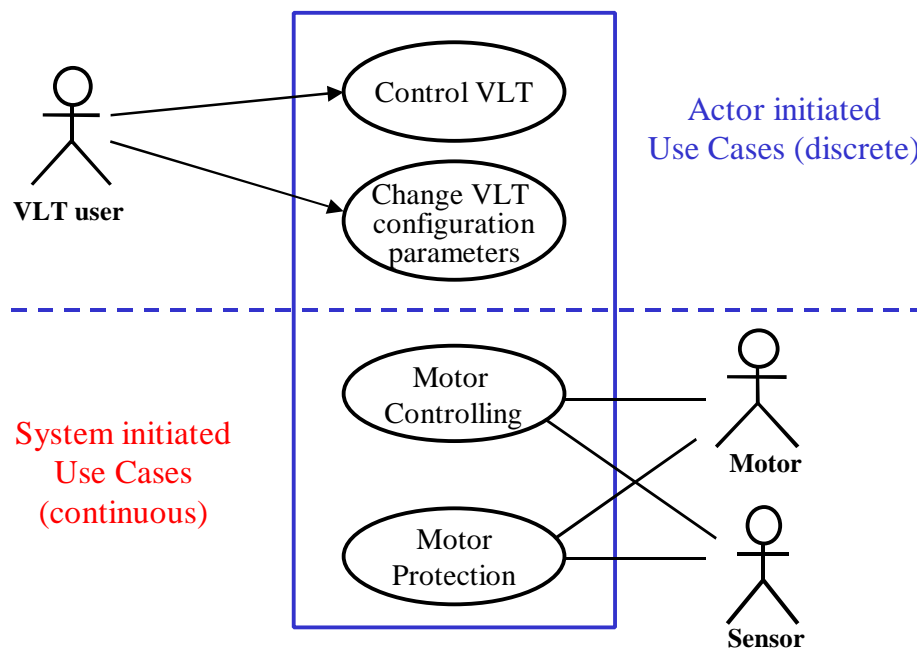


Figure 16. A part of a VLT Use Case diagram

Our experiences were that the system initiated and continuous group of Use Cases was the most difficult group to specify with the Use case technique as the technique is primarily invented for actor initiated Use Cases. Another Use Case specification problem was that the VLT is a very general device with many operation modes, which can be used, in very different applications.

The functionality specified by the Use Cases in each of these two Use Case groups' maps into classes in the corresponding parts of the two-part architectural model. This means that the two-part model functionality can be identified already in the specification phase by categorising the Use Case's in these two groups.

6.2 Task modelling

The two-part model will as a first cut lead to a task design with a minimum of two tasks. One task for controlling the discrete event controlled part and the other for taking care of the continuous processing part. In practice there will be more than one task in the continuous part as there are more than one type of processing e.g. controlling and monitoring with different timing requirements leading to two tasks in this example.

Figure 17 shows a task model example where the configuration data is encapsulated in a class realised as a “monitor” class implementing a critical region for the configuration data.

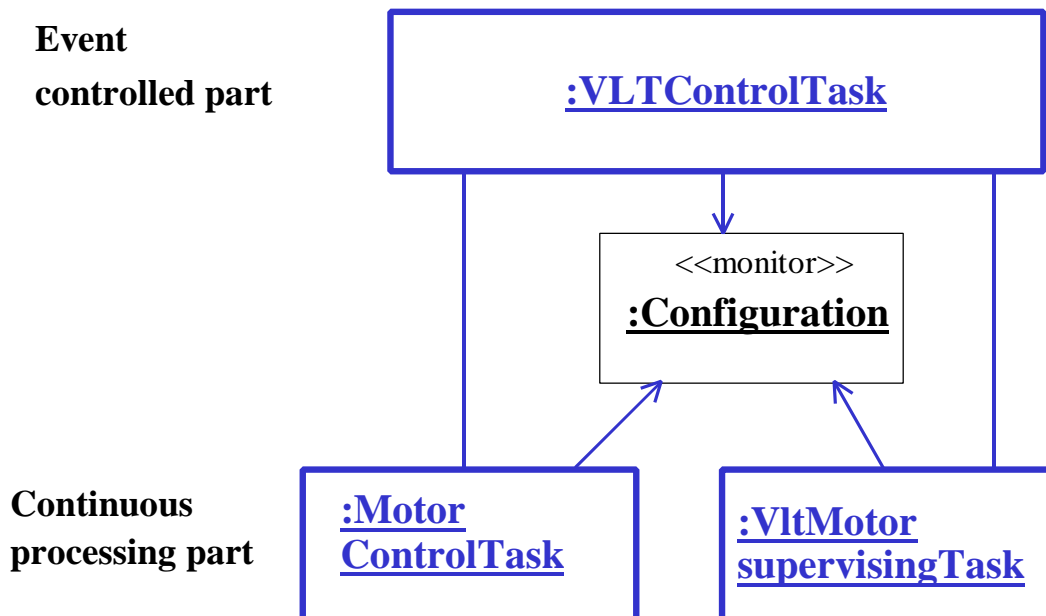


Figure 17. Task model example

6.3 Design Patterns

The knowledge and use of some of the standard design patterns described by [Gamma95], [Bushman96] and [Shaw&Garlan96] have been a great help in developing the actual object-oriented architecture. As a general guideline the continuous processing part can be designed and implemented by combining the Strategy with the Pipes and Filter design pattern. As another general guideline the discrete processing part can be designed as a state machine and implemented by combining the State with the Command pattern. The use of design patterns in the actual project has resulted in a flexible design model, which is currently evolving into a framework for frequency converters.

6.4 Other experiences with Object Technology

The OO design model is extensively based on the use of polymorph operations (i.e. C++ virtual functions), which avoid the computation of the switch arguments in a lot of switch statements at runtime. Another advantage by using polymorph operations is that the code is much easier to modify and extend without having to modify a lot of switch statements in different parts of the program. Another outcome was that the state machines for the new OO based design were considerably smaller than in previous designs based on the Ward & Mellor SA/SD Real Time method.

7. CONCLUSION

The use of Object-Oriented technology in the VLT pilot project has been very successful and demonstrated the strength of object technology. The use of OO techniques have made it possible to build a framework which currently cover two different product lines within the same design model.

The two-part architectural model has been a valuable design tool in the construction and refinements of the actual design of the VLT. We believe that the design principles shown in this paper are useful for building systems and frameworks for other kinds of industrial systems which includes continuous processing parts combined with configuration and control by external actor initiated events.

8. REFERENCES

[Bushmann96]:

A System of Patterns: Patterns Oriented Software Architecture
Frank Bushmann, Regine Meunier, Hans Rohnert, Peter Sommerlad,
Michael Stahl
John Wiley & Sons, 1996

[COT]:

The Centre for Object Technology (COT) is a three year project concerned with research, application and implementation of object technology in Danish companies. The project is financially supported by The Danish National Centre for IT-Research (CIT) and the Danish Ministry of Industry. (www.cit.dk/COT/)

[Gamma95]:

Design Patterns: Elements of Reusable Software
Eric Gamma, Richard Helms, Ralph Johnson, John Vlissides
Addison Wesley Longhamm, 1995

[Jacobson92]:

Object-Oriented Software Engineering – A Use Case Driven Approach
Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Övergaard
Addison-Wesley, Revised fourth printing 1993.

[Shaw&Garlan96]:

Software Architecture: Perspective of an Emerging Discipline
Mary Shaw and David Garlan
Prentice-Hall, 1996

[UML97]:

Unified Modeling Language (UML)
Grady Booch, Ivar Jacobson, James Rumbaugh, et. al.
An industry standard – standardized in November 1997 by
Object Management Group (OMG). (www.omg.org)