Designing Event-Controlled Continuous Processing Systems
Class 325

by

Hans Peter Jepsen, Danfoss Drives

and

Finn Overgaard Hansen, Engineering College of Aarhus

hans_peter_jepsen@Danfoss.com
foh@e.iha.dk

The main ideas of this class

- To present the

  two-part architectural model

  as a basis for implementing

  event-controlled continuous processing systems

- To outline an

  object-oriented design of this architectural model

  where “design patterns” have been helpful in the design

- To give

  some method hints

Results of a pilot project on the OO-development of a frequency converter in the Danish research project COT - Centre for Object Technology

What is an "*Event-Controlled Continuous Processing System*" ?

- A system
  - which carries out continuous signal processing
  - which reacts on events that impacts and reconfigures the signal processing
- The signal processing
  - can be carried out by software and/or hardware,
  - can be distributed on multiple processors

When carried out in software or digital hardware, signal processing is periodic or discrete-time

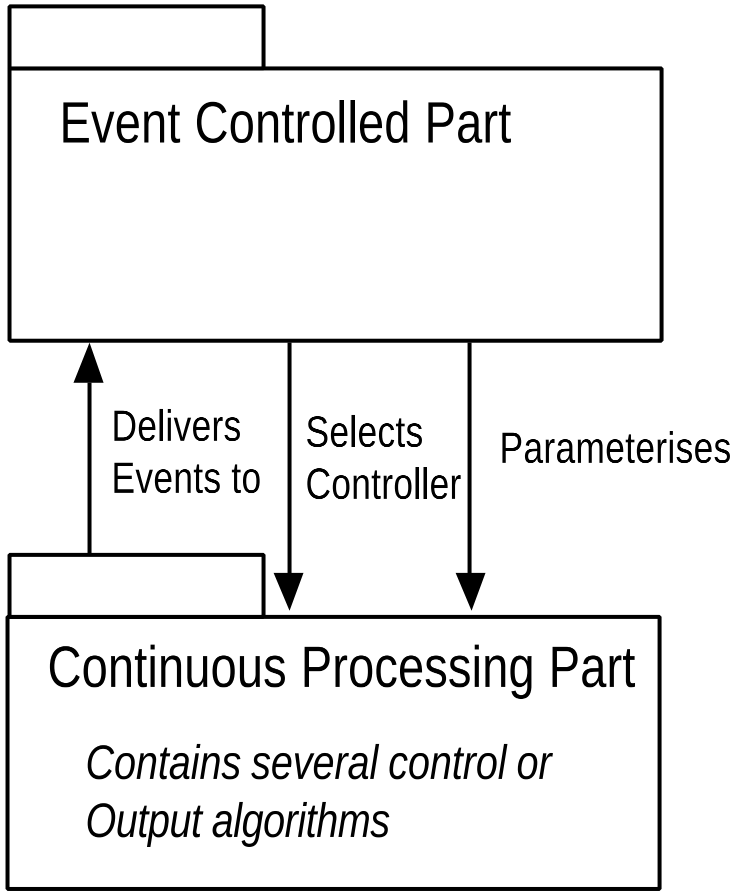
Examples of Event-Controlled Continuous Processing Systems

- Examples can be found in several fields, e.g
  - Measurement instruments (e.g. flowmeters)
  - Process control (e.g. frequency converters)
  - Consumer electronic (e.g. CD-players)

The frequency converter
- as an example

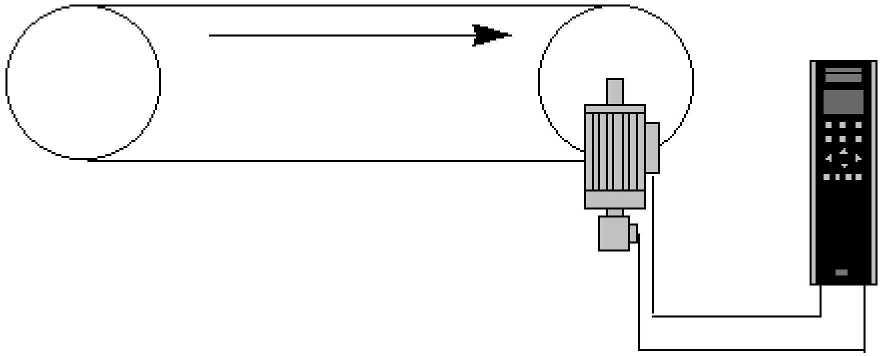
- A *frequency converter* is a device used to control a three-phase induction motor
  - so that the motor speed or motor torque matches the need of a given application
- Often called a “drive”
  - which explains the company name “Danfoss Drives”
- VLT® is the trademark for Danfoss frequency converters

The two-part architectural model



- The event-controlled part:
  - Responsibilities: event handling, configuration
  - Selects controller in cont. proc. part
- The continuous processing part:
  - Responsibility: continuous data processing
  - Delivers events (indications) to event-controlled part, e.g. ”Spinning motor caught”

Controlling a conveyer belt - example of speed control



Goal:

– To control the speed of the conveyer belt

Constraints:

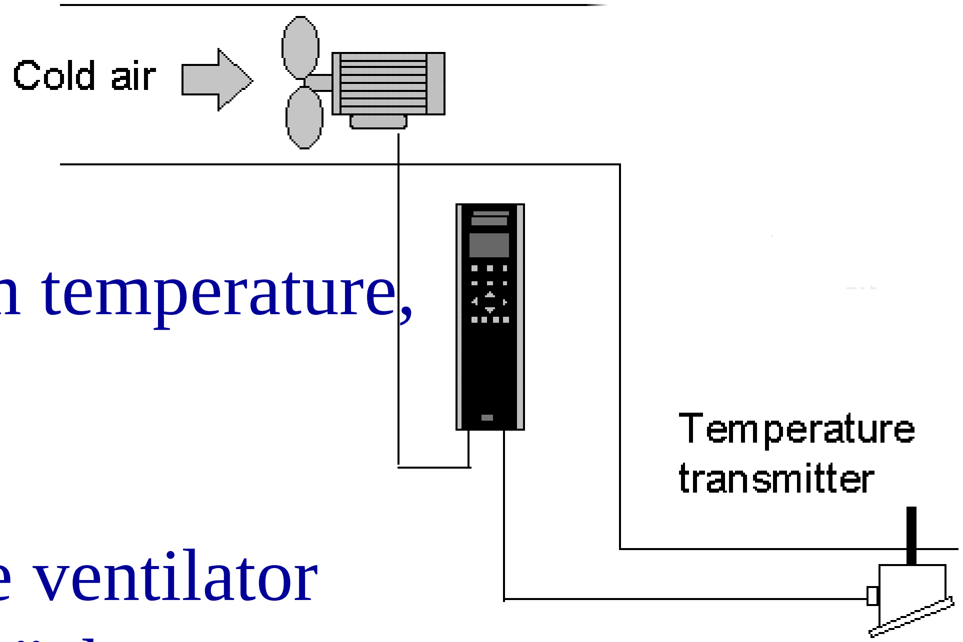
– Changes in load may not change the speed

– Changes in speed (e.g. start and stop) must

  – occur in a controlled manner and

  – be “smooth”

Controlling a fan
- example of process control



Goal:

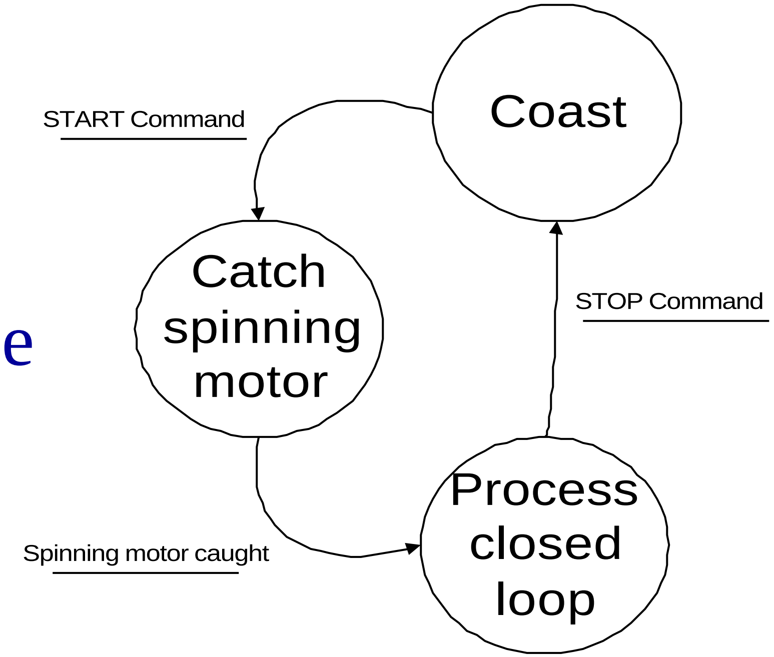
- To maintain a desired room temperature, e.g. in an airport building

Constraints:

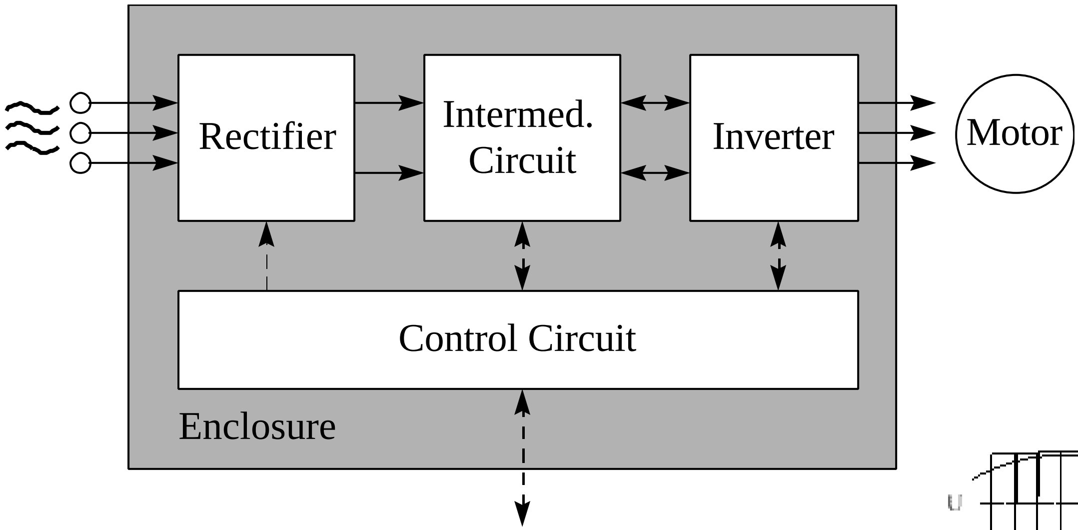
- Before applying power, the ventilator must be stopped or "caught", because the ventilator is often "windmilling" when the motor is not powered

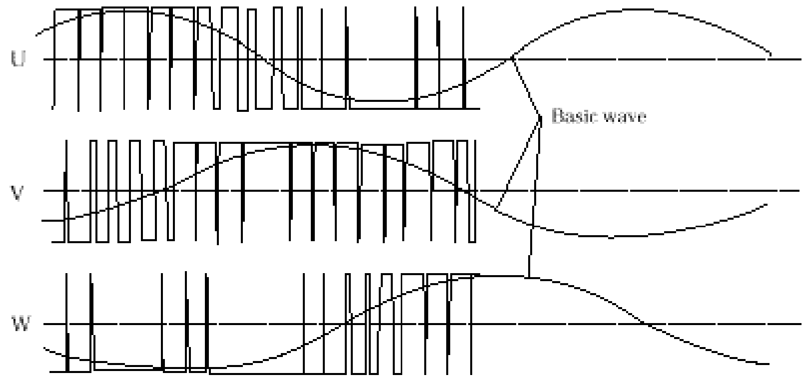
More to the Fan Control

- Three modes of steady-state operation
  - Coast (fan is ”wind milling”)
  - Catch spinning motor
  - Closed loop feedback control
- Events control shift between these
  - Start command
  - Spinning motor caught
  - Stop command
- Demand: ”Bumpless transfer”



The working of a frequency converter



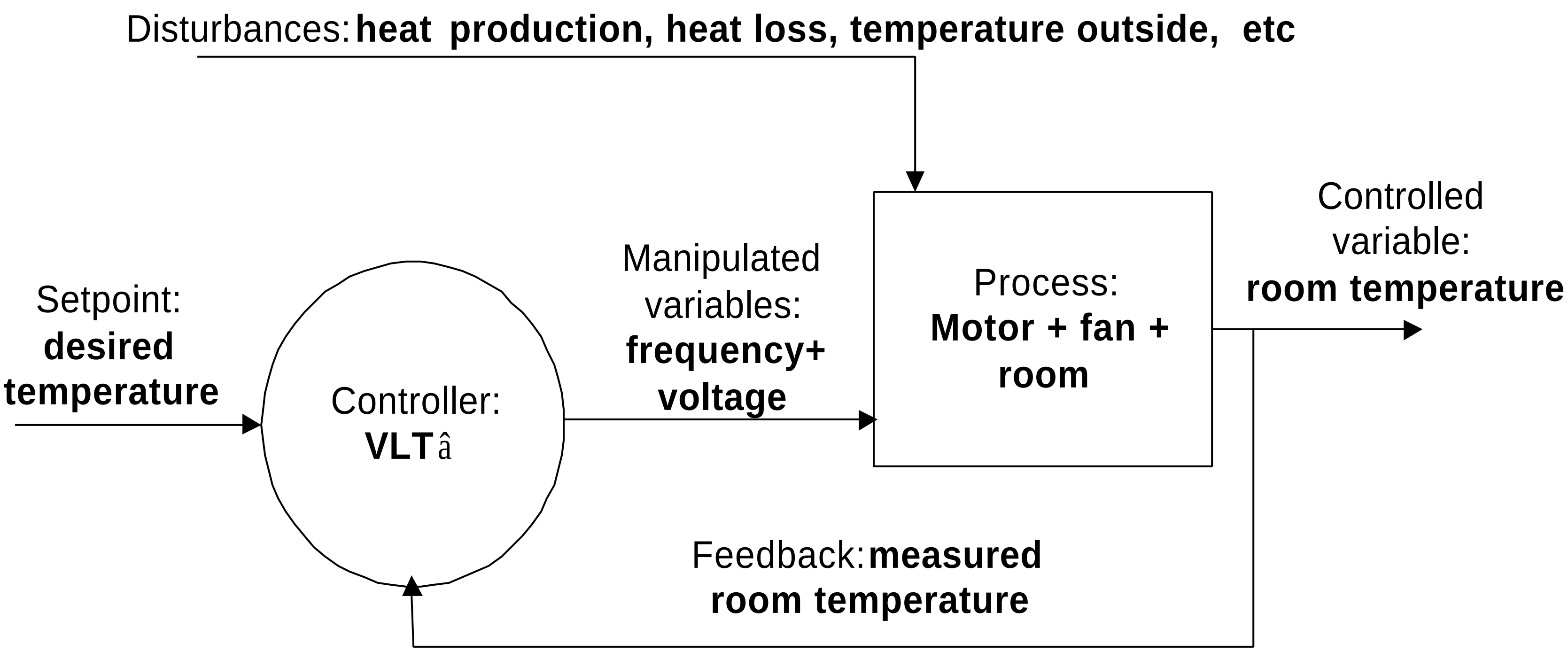


c) Time sequence of the control signals for three inverter phases – U, V, W

“Architectural style: Process Control”

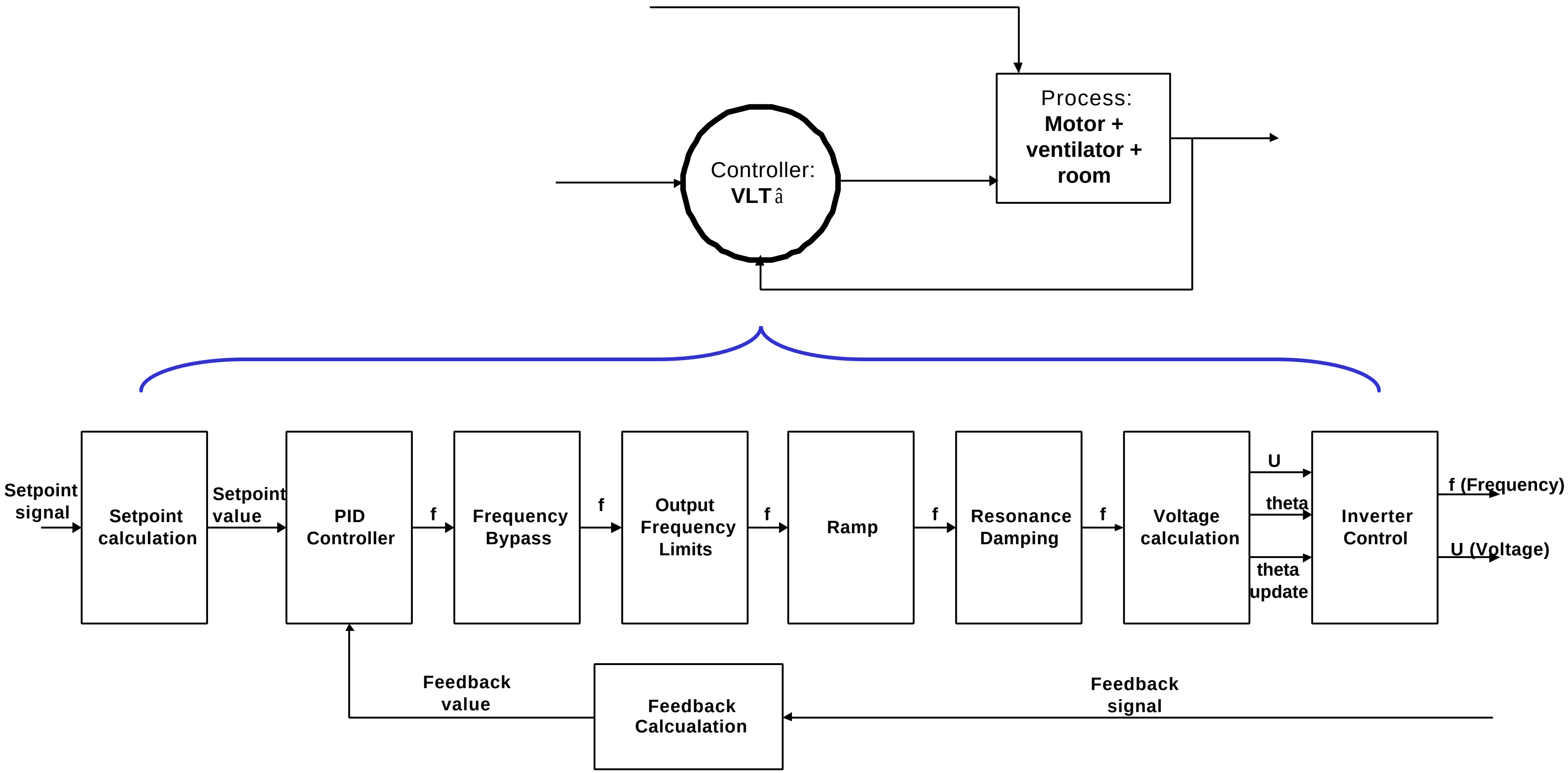
- “Architectural style” - a pattern for the architecture of a group of systems
- The “Process Control” architectural style can be used with advantage in connection with continuous control of a process
- The style is presented by applying it to the frequency converter examples
- Reference: [Shaw&Garlan96]

Ventilator control as process control



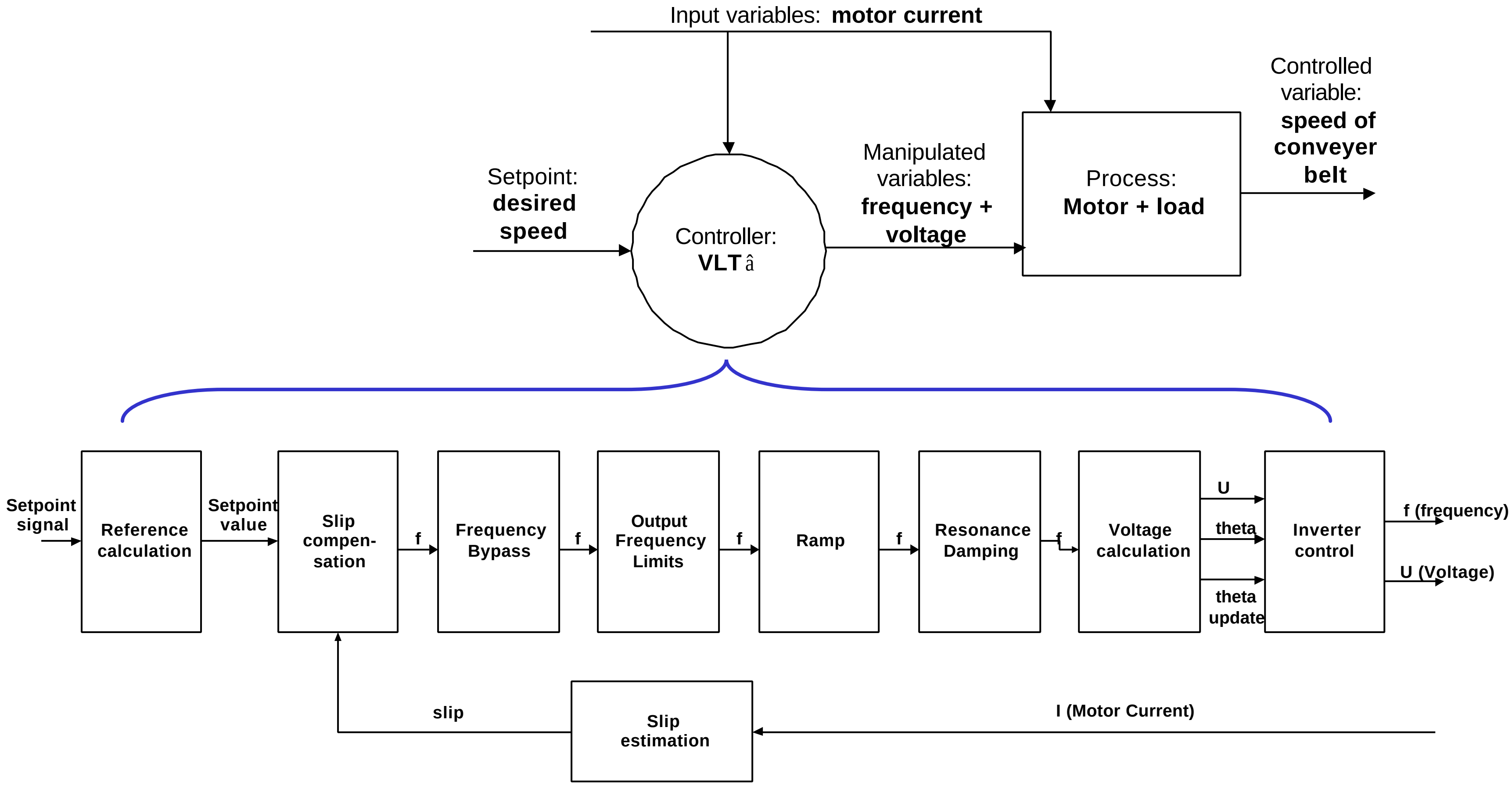
Example of "closed-loop feedback" control

The inside of the controller is signal processing



The "Process Closed Loop" controller in a VLT®

The conveyer belt again



The "Speed Open Loop" controller in a VLT® - also called "sensorless speed control". Example of a feedforward control system.

Continuous signal processing - implemented in software

Characteristics:

- Periodic instead of continuous
  - normally initiated by a periodic interrupt
- Input signals are sampled
- Period length is an important system parameter
- Dataflow architecture
- Thorough treatment in control theory
  - but often lacks guidelines for sensible SW-implementations

A frequency converter must also react to events

- Events can be:
  - commands, e.g: Start, Stop, Change-Setup
  - change of a “parameter” value
  - new set point value received from a “process-local-area-network”
- Sources can be
  - digital terminals
  - keypads
  - telegrams from the “process-local-area-network”

The treatment of events

Characteristics:

- The arrival of events is often independent of and asynchronous to signal processing
  - normally via an interrupt
- The reaction is very often determined by finite state machines

The event treatment impacts signal processing

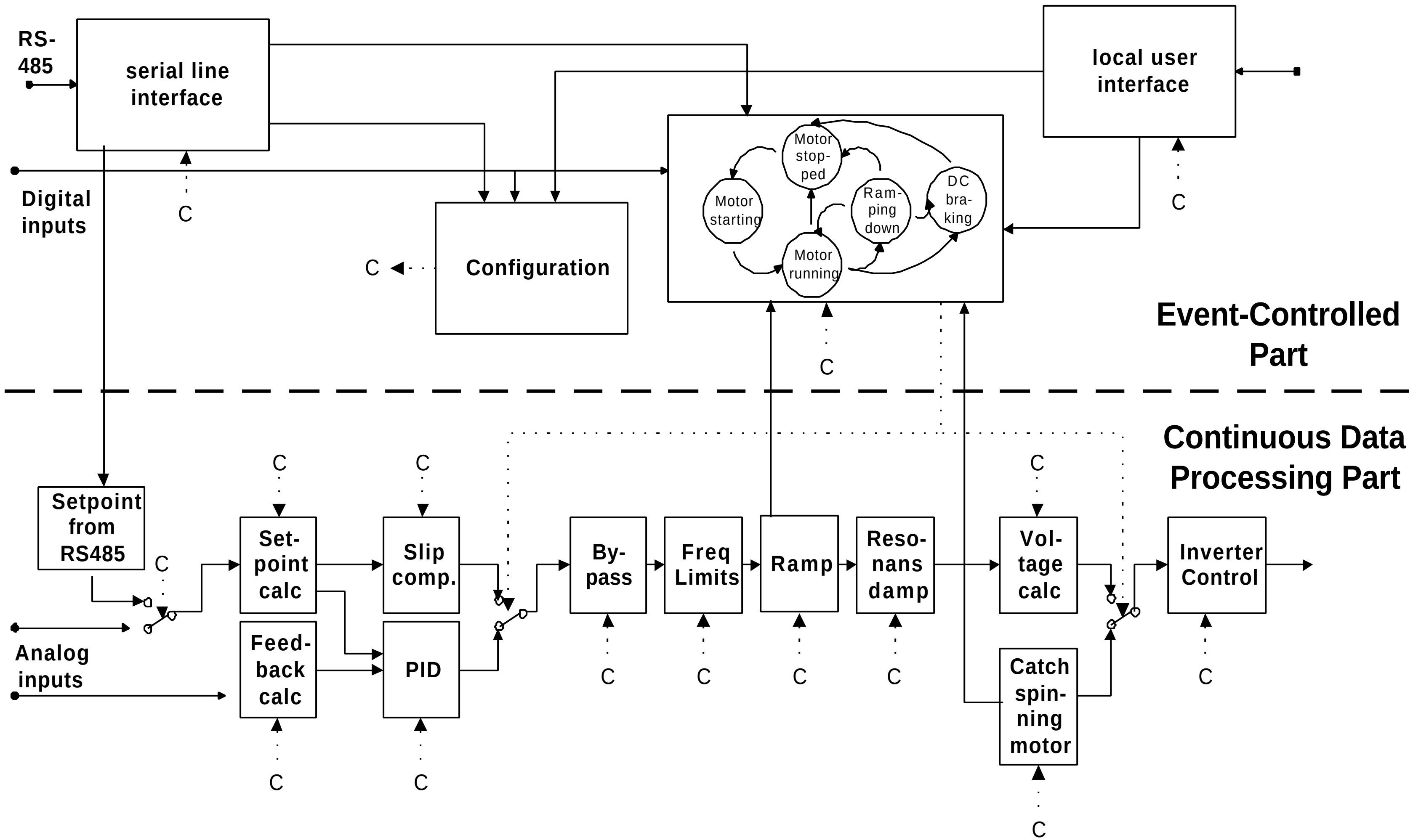
- Most signal processing “block” has configuration-parameters
  - e.g. ramp: ramp time, type (linear or S-ramp)
- Most commands and configuration-parameters result in a change in the signal path
  - Start, Stop, Change-Set-up
  - Change from Speed-Open-Loop to Process-Closed-Loop

The signal processing part can produce events

Examples:

- The setpoint-signal “disappears”
  - the motor must be stopped or the controller replaced
- The motor speed reaches 0 rpm
  - the motor control mode must be set to “Stopped”
- Feedback outside user determined limits
  - a “Warning” must be submitted

The two-part architecture (very simplified)



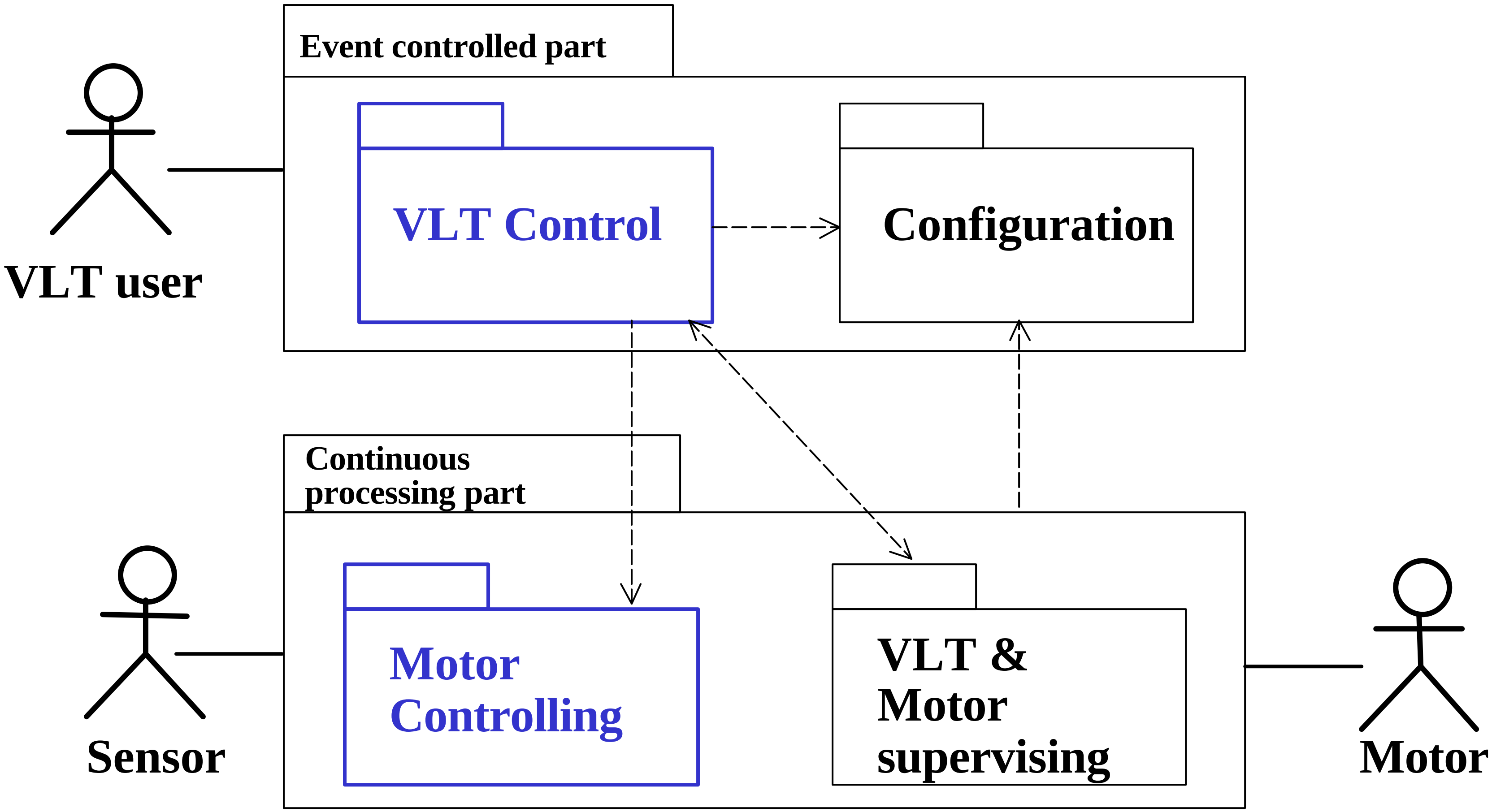
Advantages of the two-part architectural view

- The two parts of the software have different demands and constraints
  - i.e. the questions, that must be answered, and the approach is different
- The software can be implemented so that the signal processing path is only set up, when it needs to be changed
  - in our former approach the path was set up during every period
- The period length of the different blocks does not have to be identical

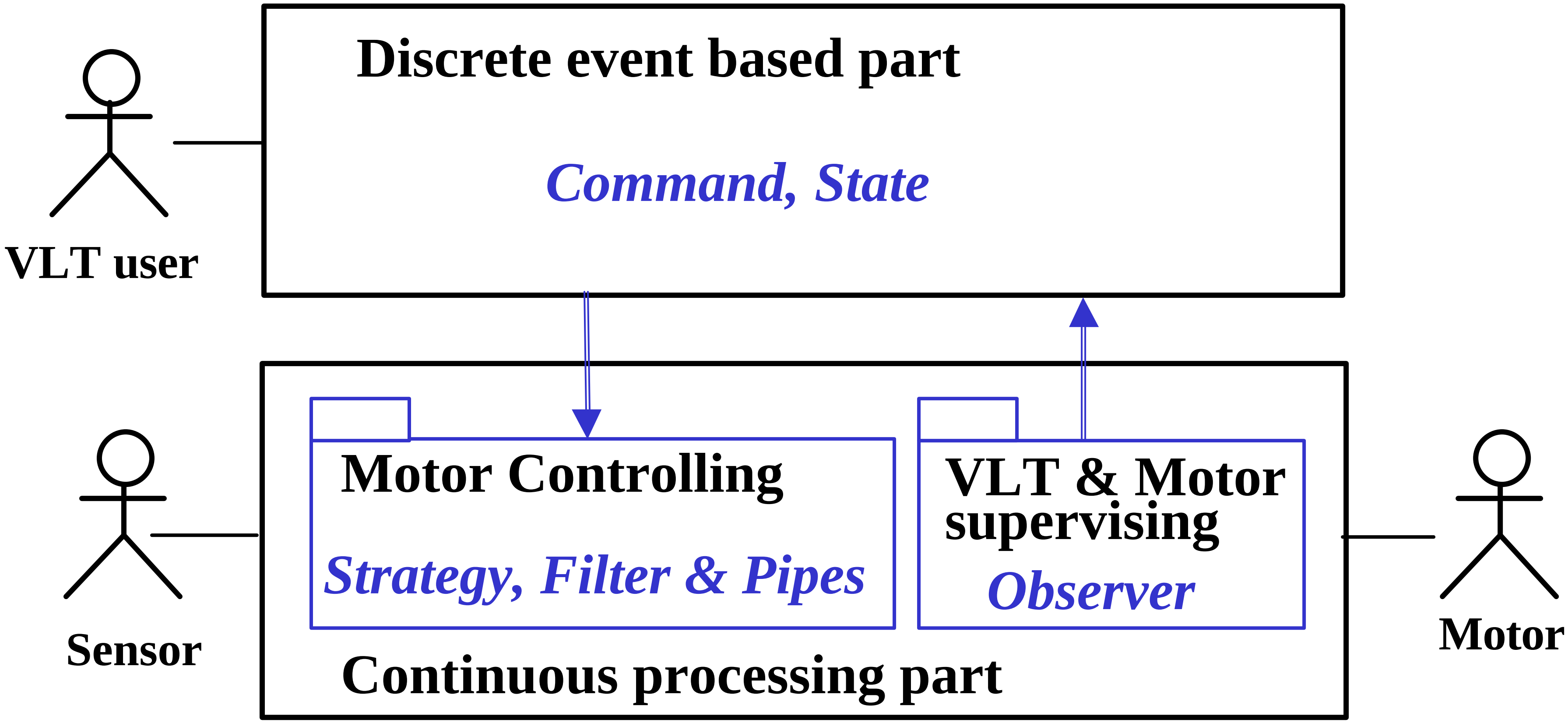
The applicability of the two part architectural view

- Assumed to be applicable in many embedded systems
  - The signal processing can be distributed on multiple CPU's
  - The signal processing "blocks" can be moved between HW and SW

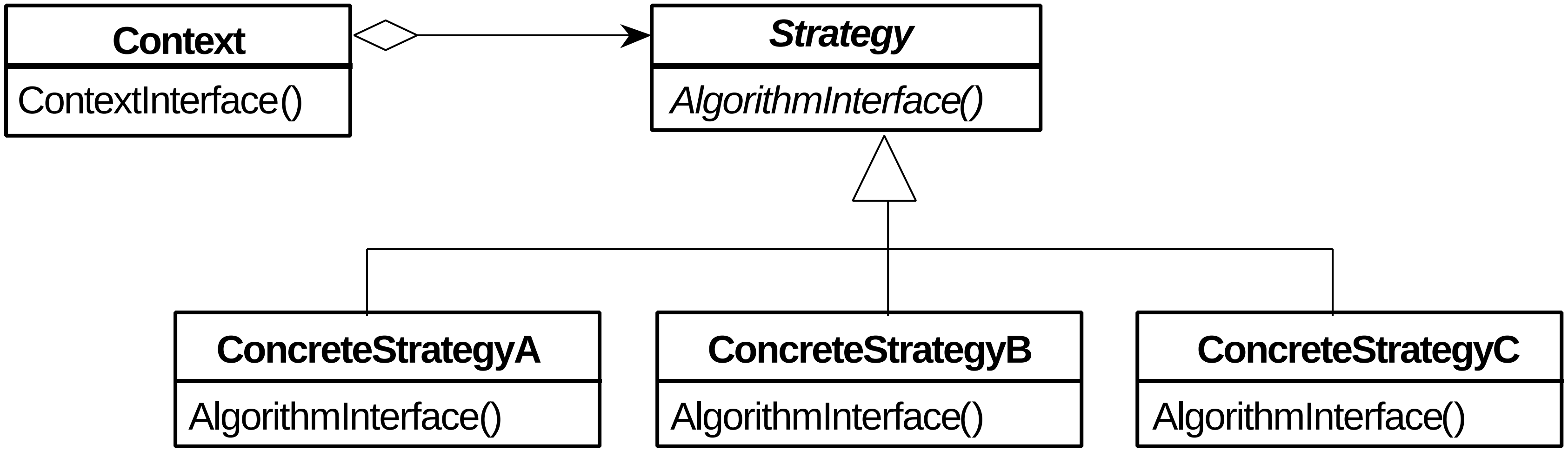
Two-part model for a frequency transformer (VLT)



Design Patterns in the two-part architectural model

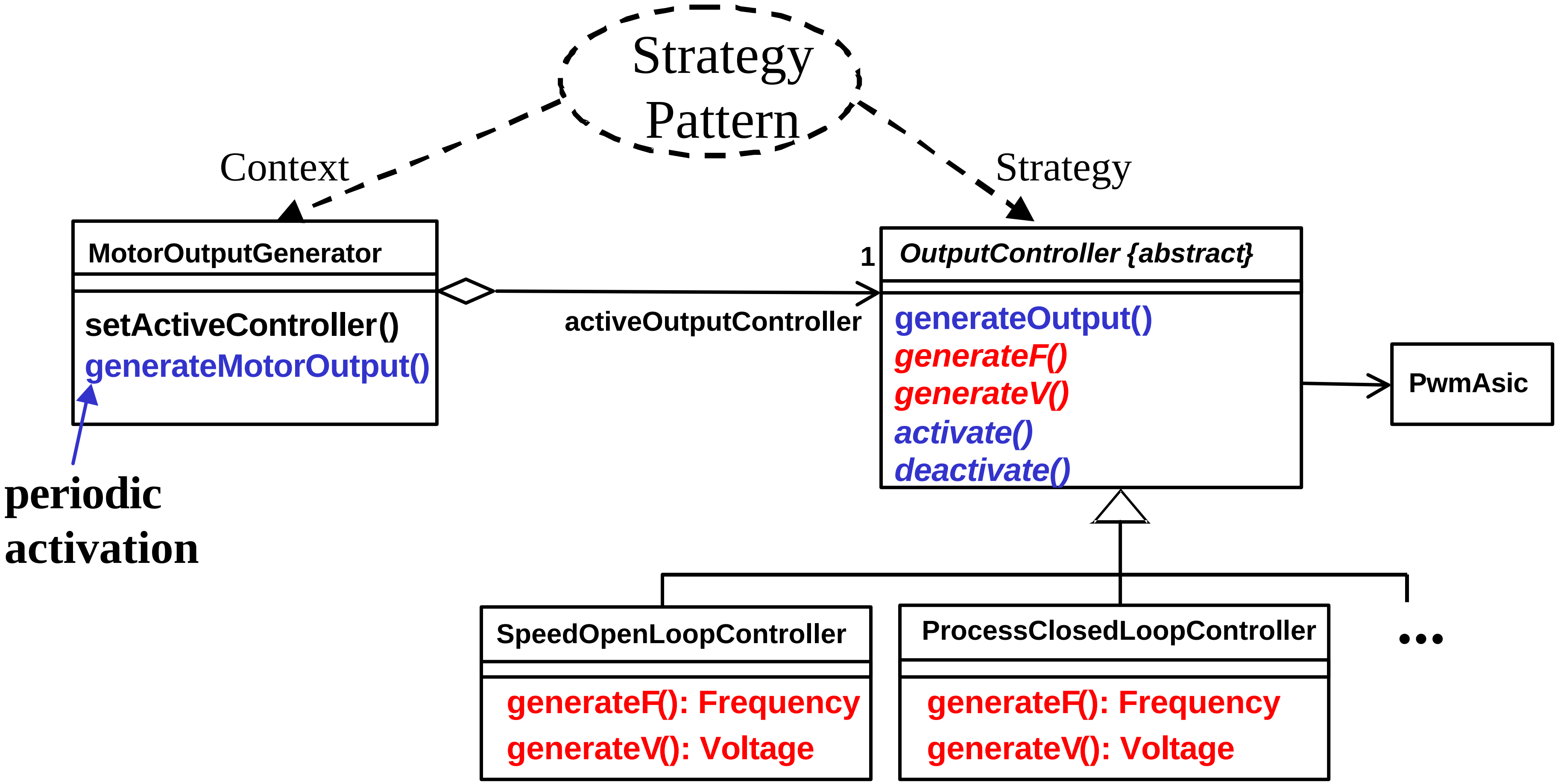


Structure for Strategy Pattern



Ref. [Gamma95]

Class diagram showing realisation of Strategy Pattern

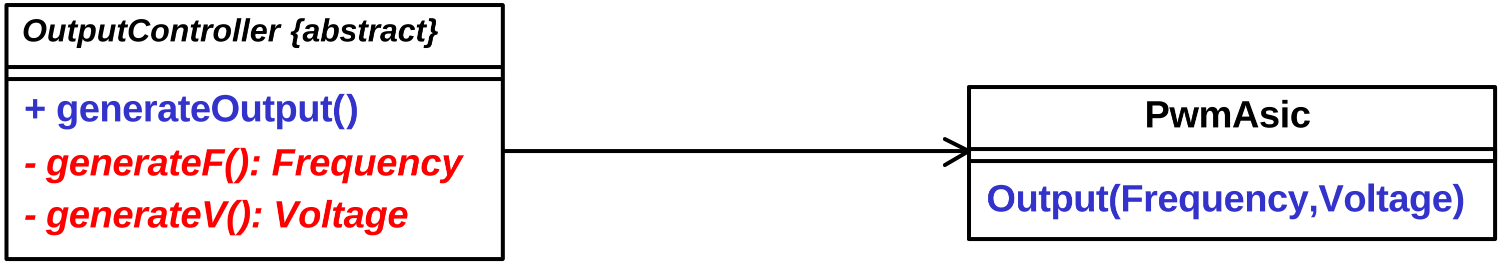


C++ code example for ‘generateMotorOutput()’


```
OutputController *theActiveOutputController;


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

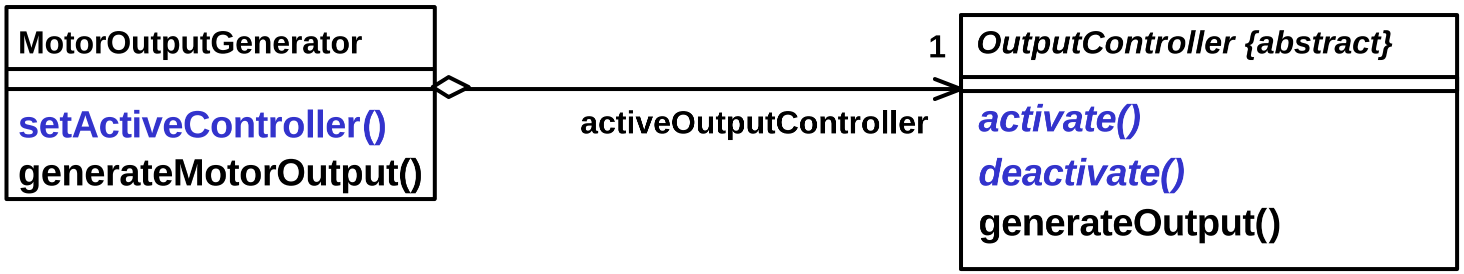
C++ code example for ‘generateOutput()’




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

generateOutput() is a Template Method according to [Gamma95]

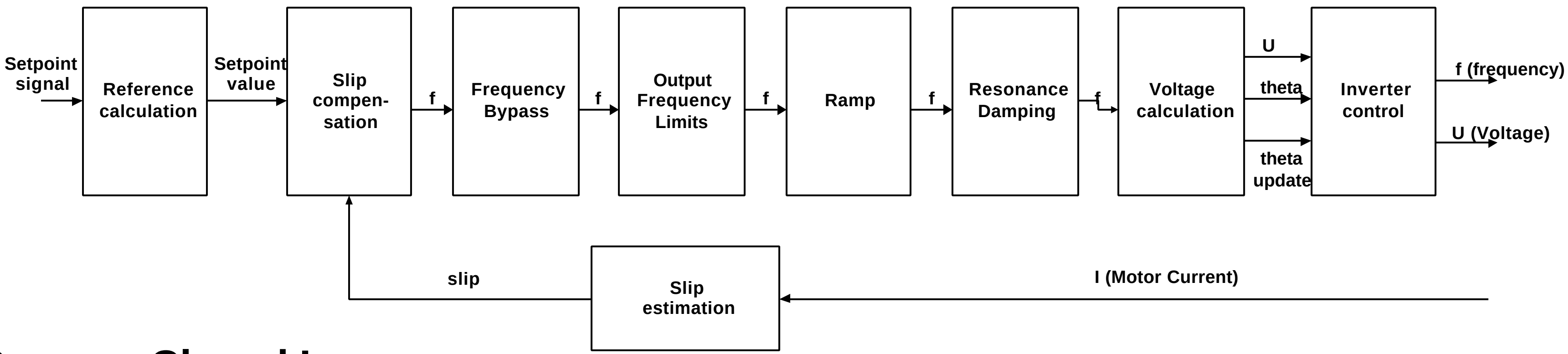
C++ code example for 'setActiveController()'



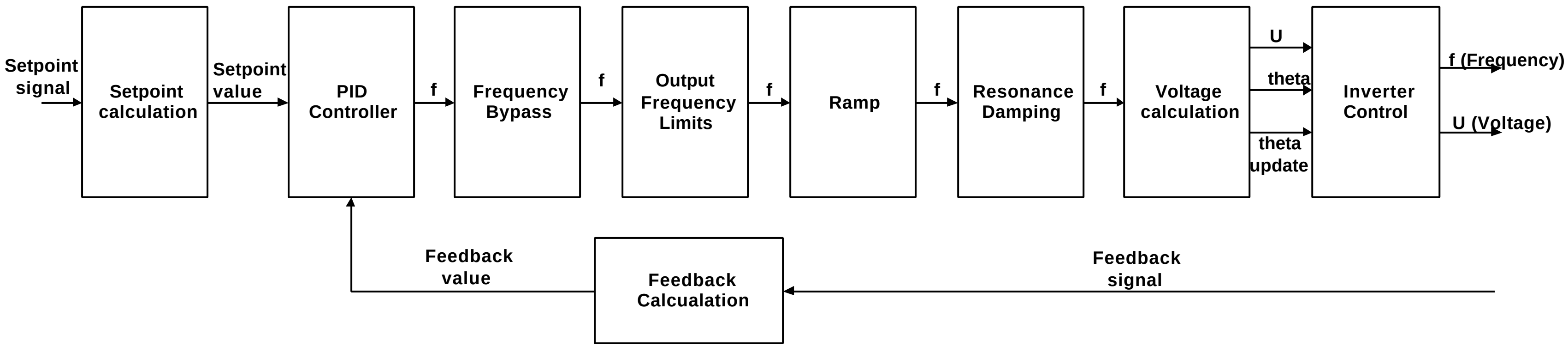

```
MotorOutputGenerator::setActiveController
                                   (OutputController: newController)
{
    controllerInfo= theActiveOutputController->deactivate();
    theActiveOutputController= newController;
    theActiveOutputController->activate(controllerInfo);
}
```

Block diagram for two different application modes

Speed Open loop:



Process Closed Loop:



Pipes and Filters Pattern classes

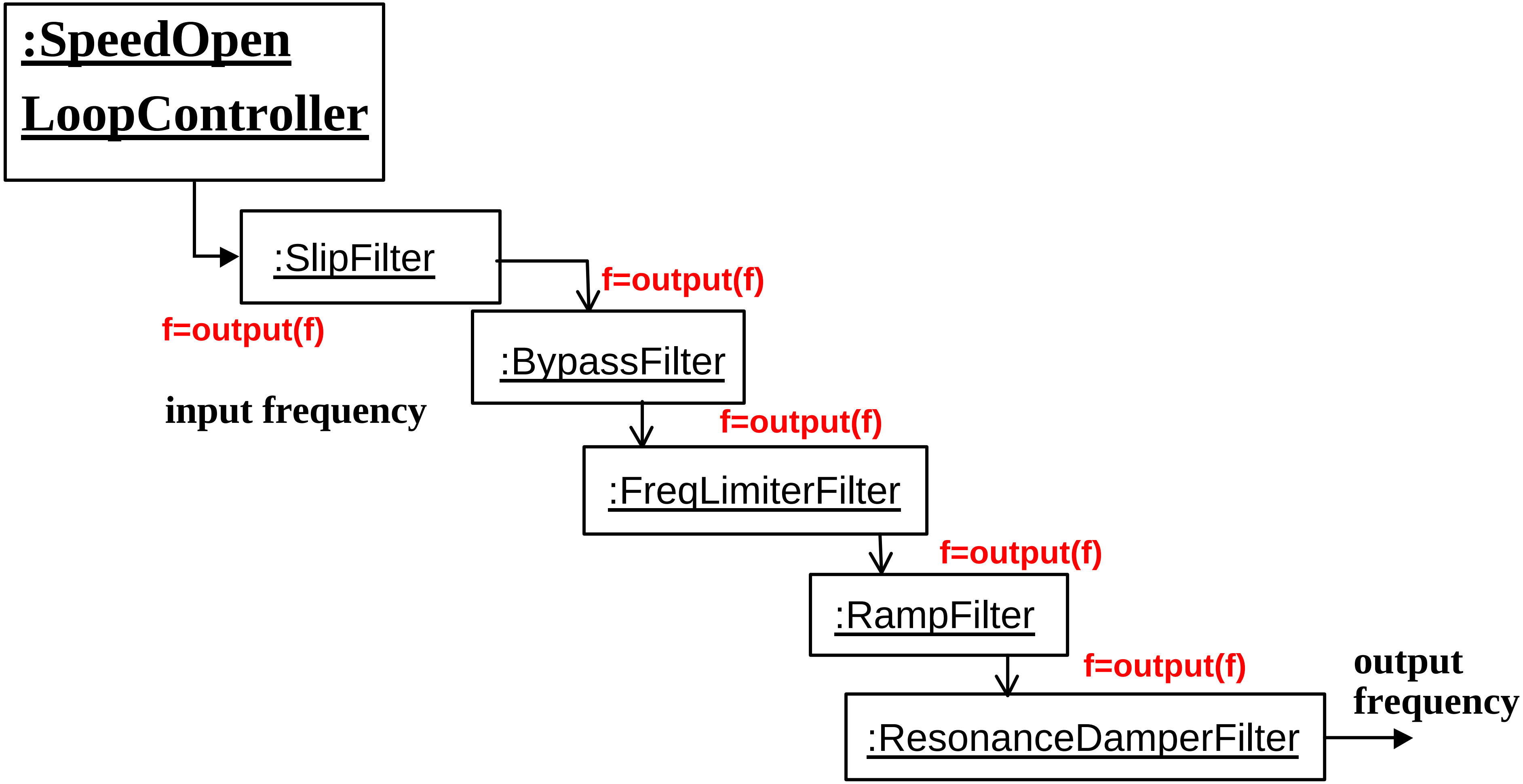
Class Filter	
Responsibility	***Collaborators***
Get input Perform function Set output	Pipe

Class Pipe	
Responsibility	***Collaborators***
Transfer data Buffer data Sync. filters	Data Source Data Sink Filter

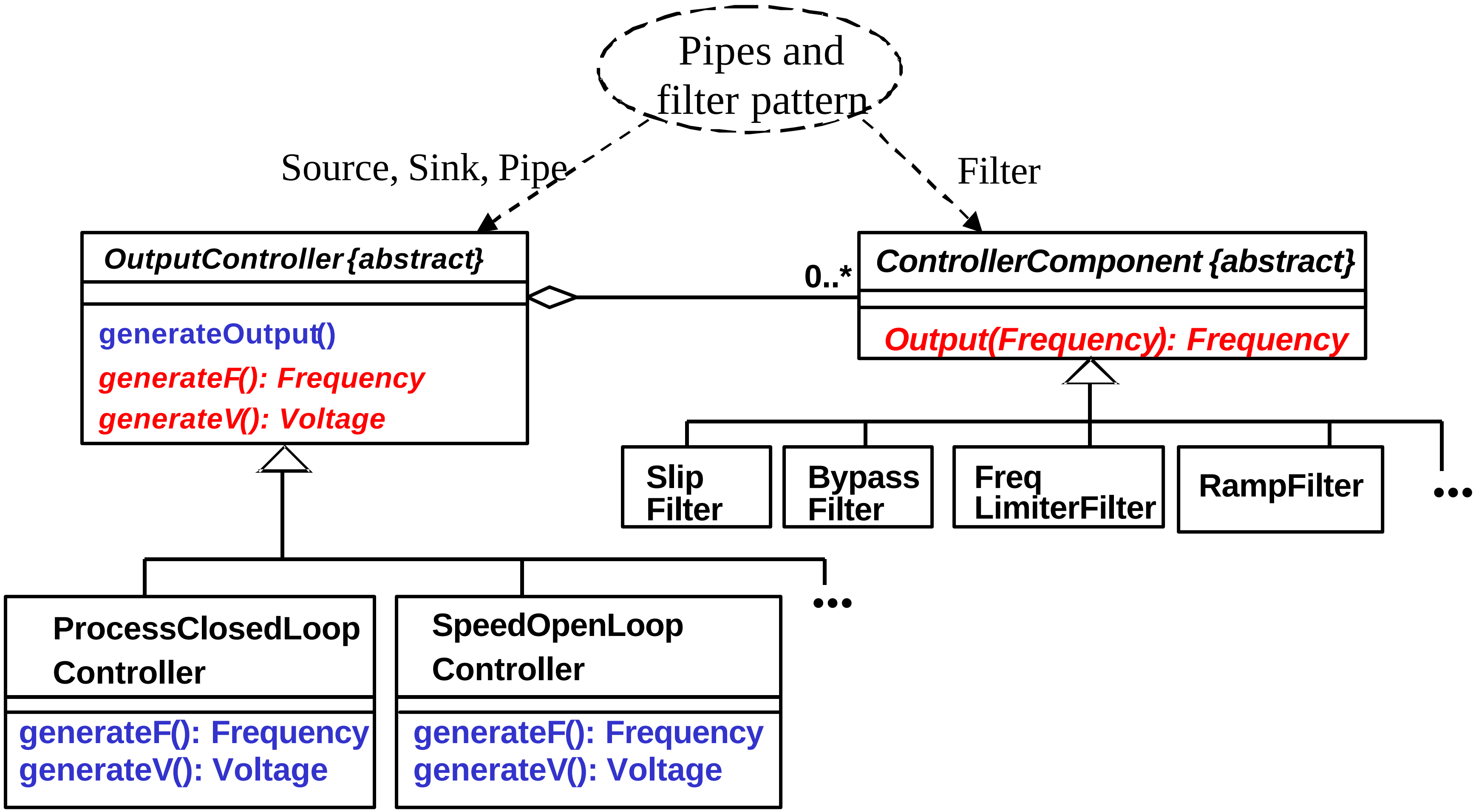
Class Data Source	
Responsibility	***Collaborators***
Deliver input to processing pipeline	Pipe

Class Data Sink	
Responsibility	***Collaborators***
Consumes output	Pipe

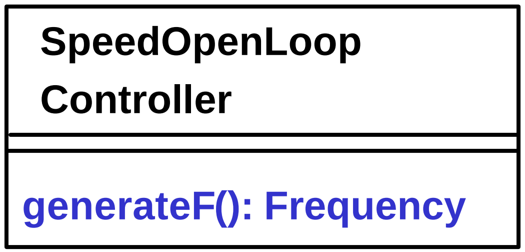
Object diagram for SpeedOpenLoopController



Class diagram for Pipes and Filter pattern

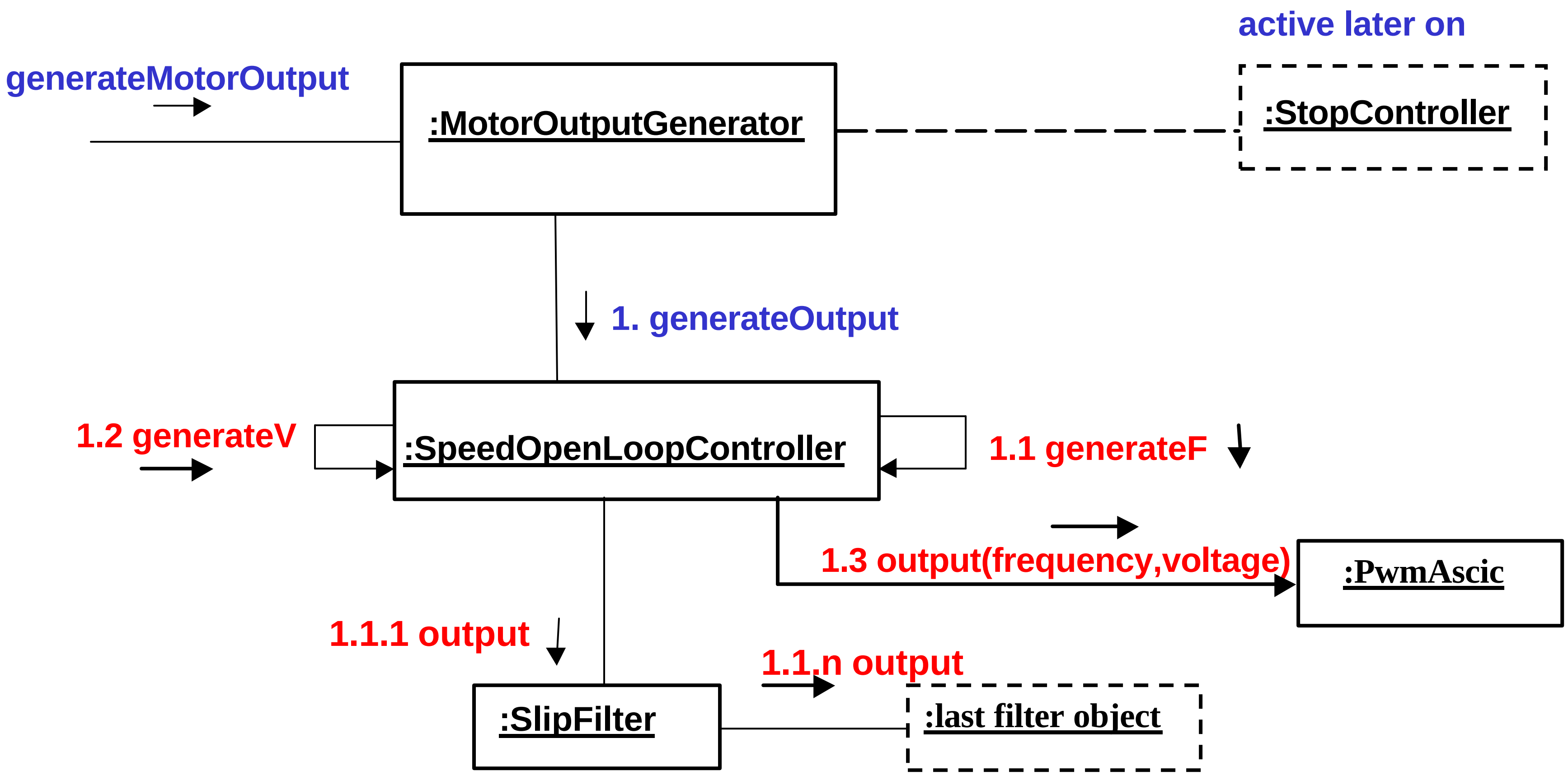


C++ code example for ‘generateF()’

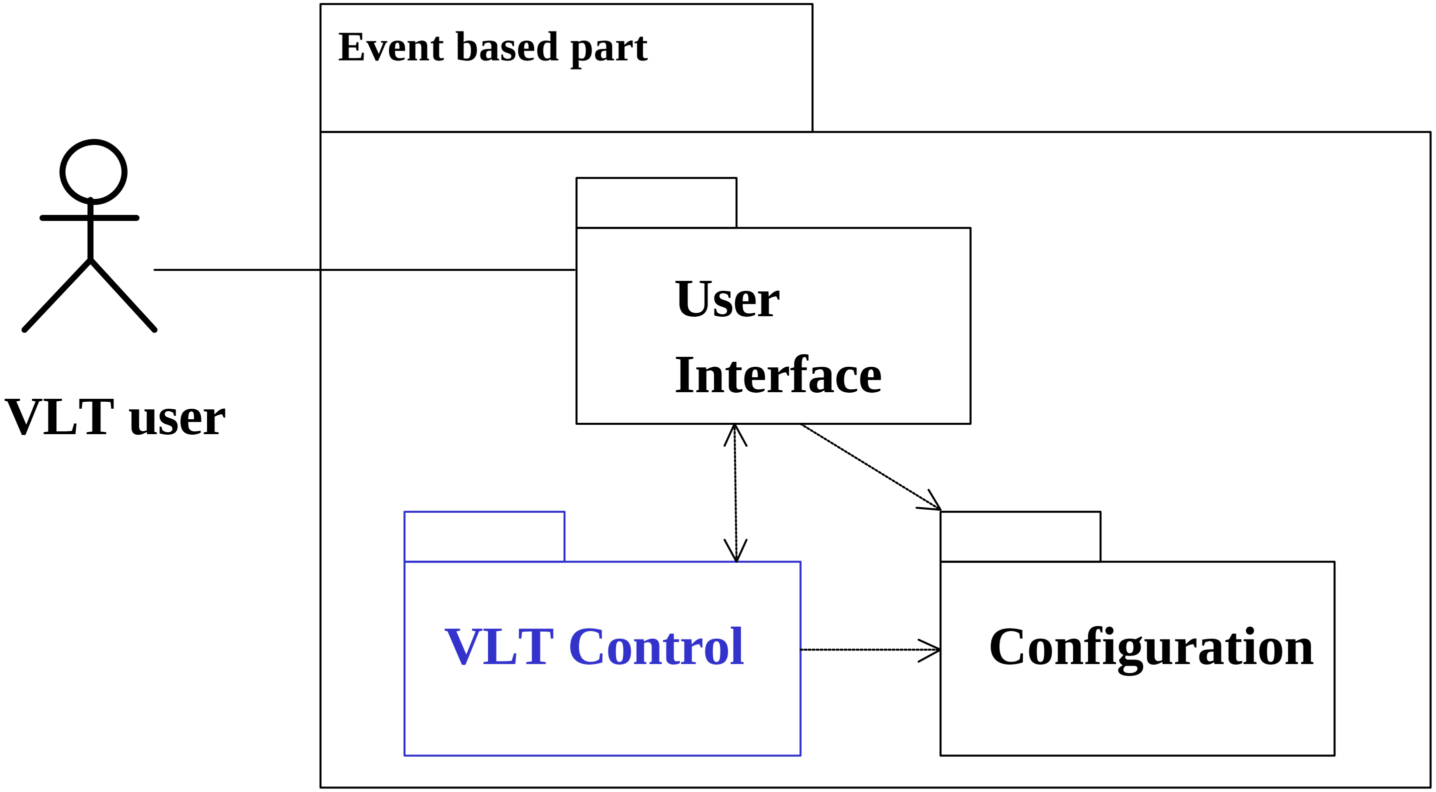



```
Frequency SpeedOpenLoopController::generateF()
{
        …..
        frequency = theSlipFilter->output(frequency);
        frequency = theBypassFilter->output(frequency);
        frequency = theFreqLimiterFilter->output(frequency);
        frequency = theRampFilter->output(frequency);
        frequency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

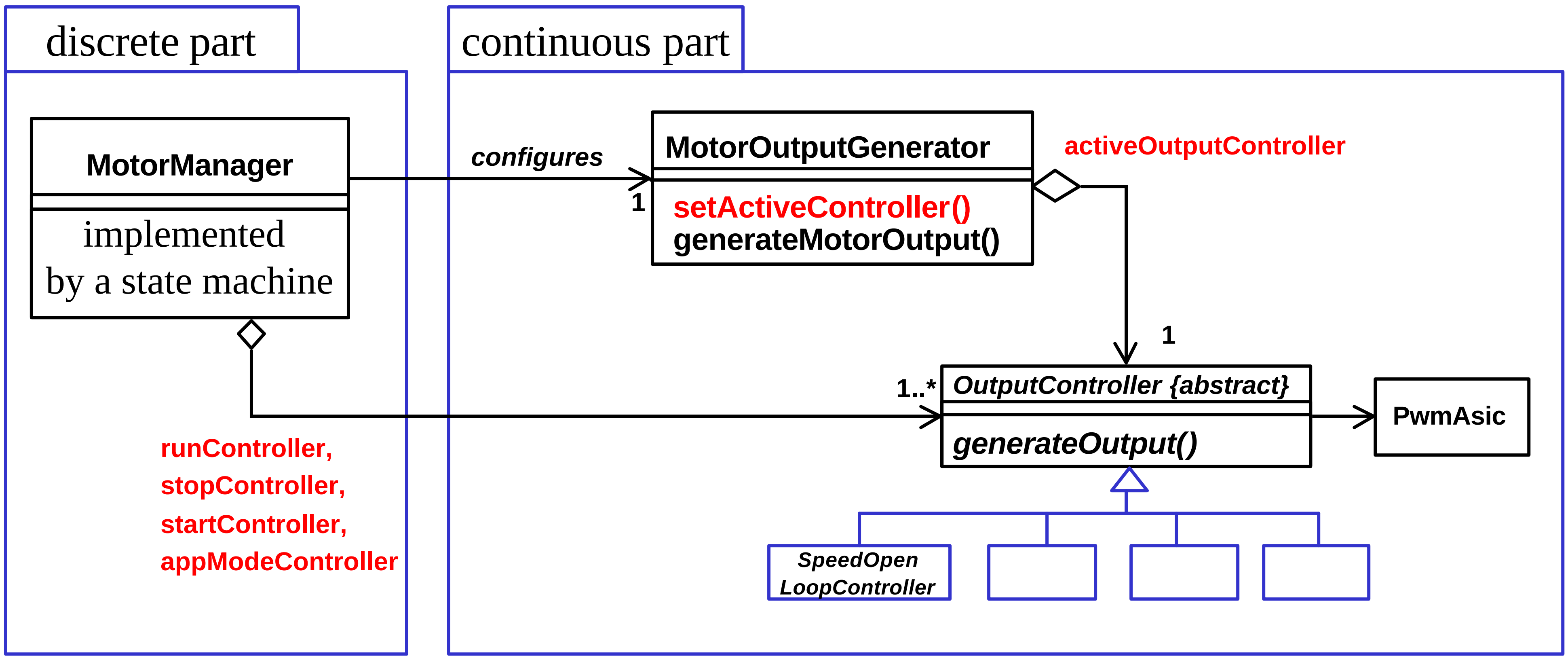
Object ”collaboration” diagram for ‘generateMotorOutput’



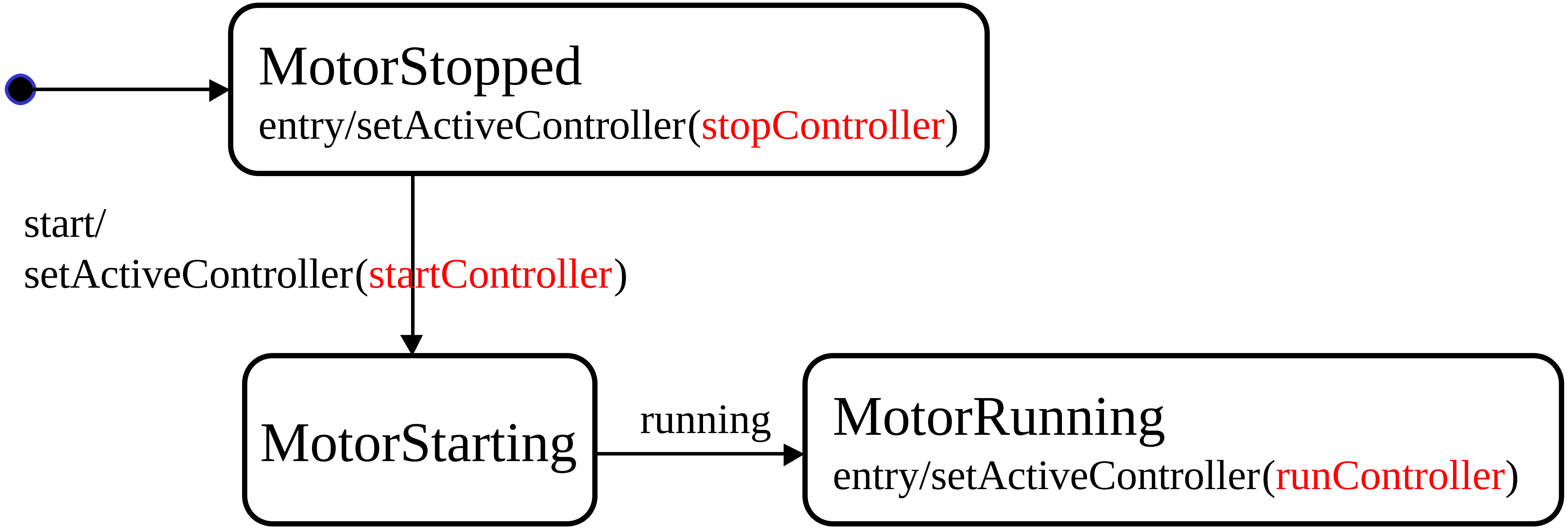
Outline of the discrete event based part



Configuration of motor output generator



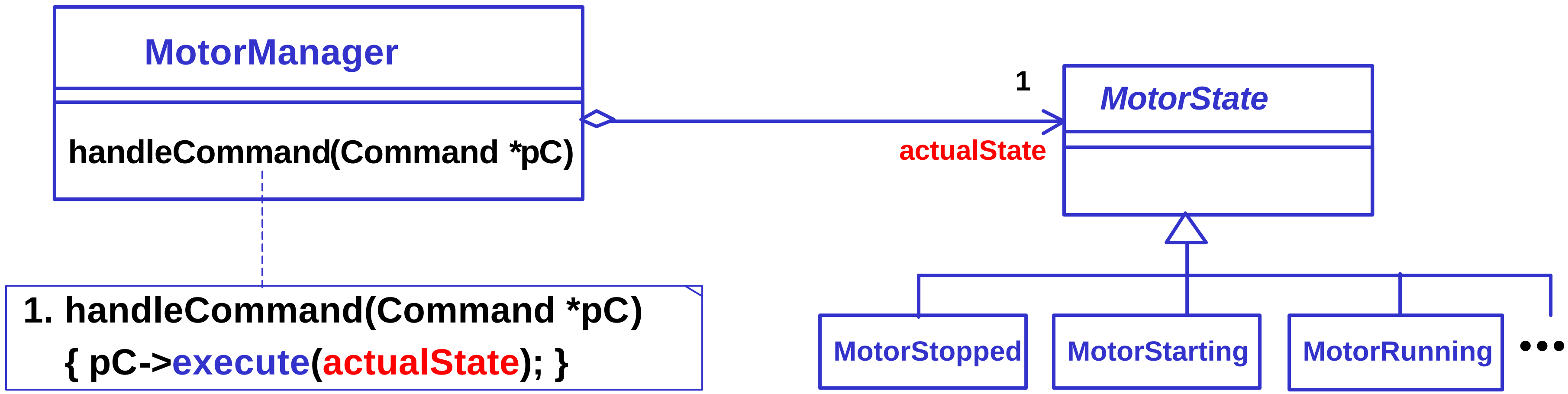
Part of the state machine for the MotorManger class



Initialized in another part of the program:

runController= theConfiguration->AppModeController())

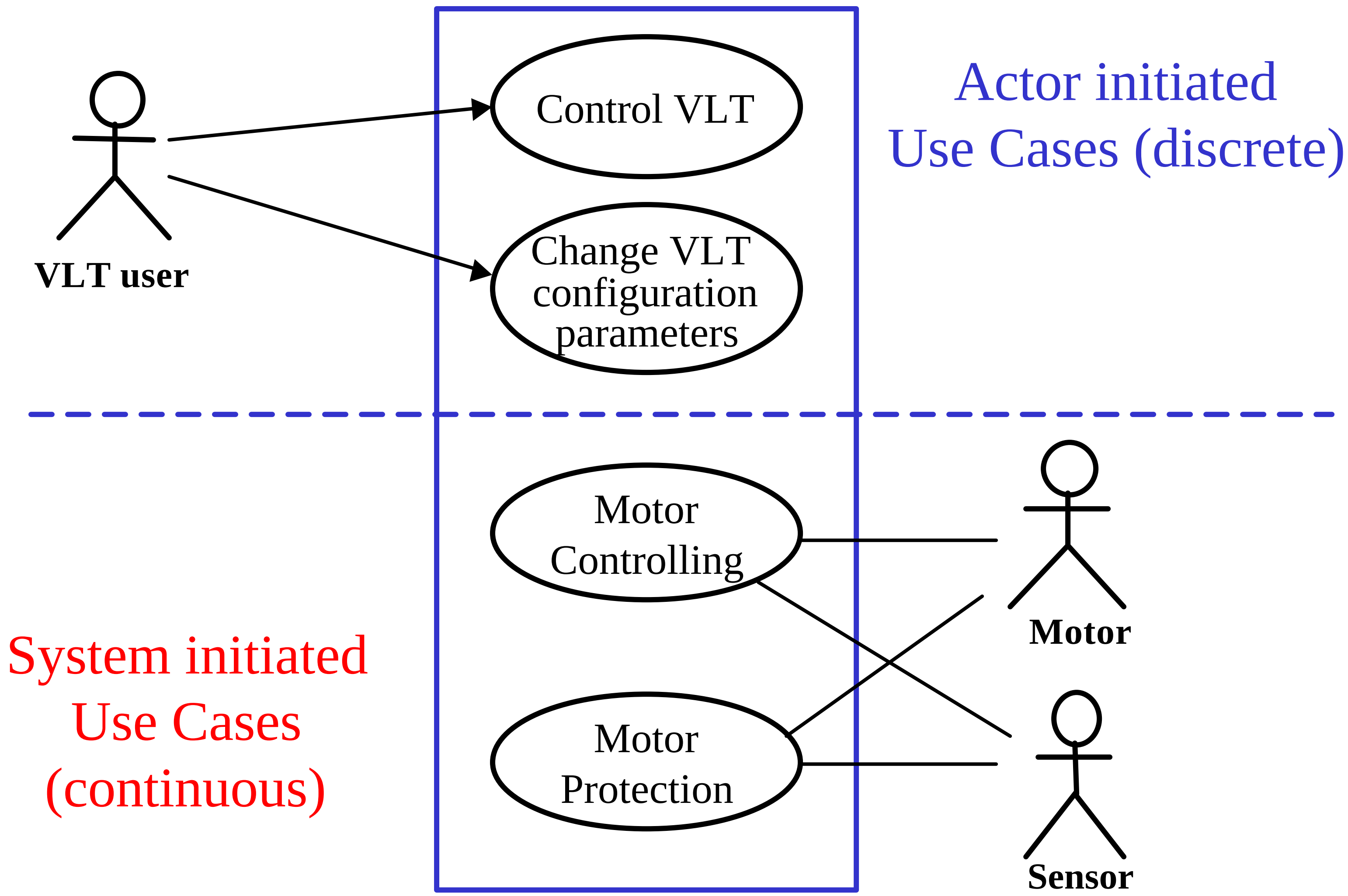
Outline of Command + State pattern



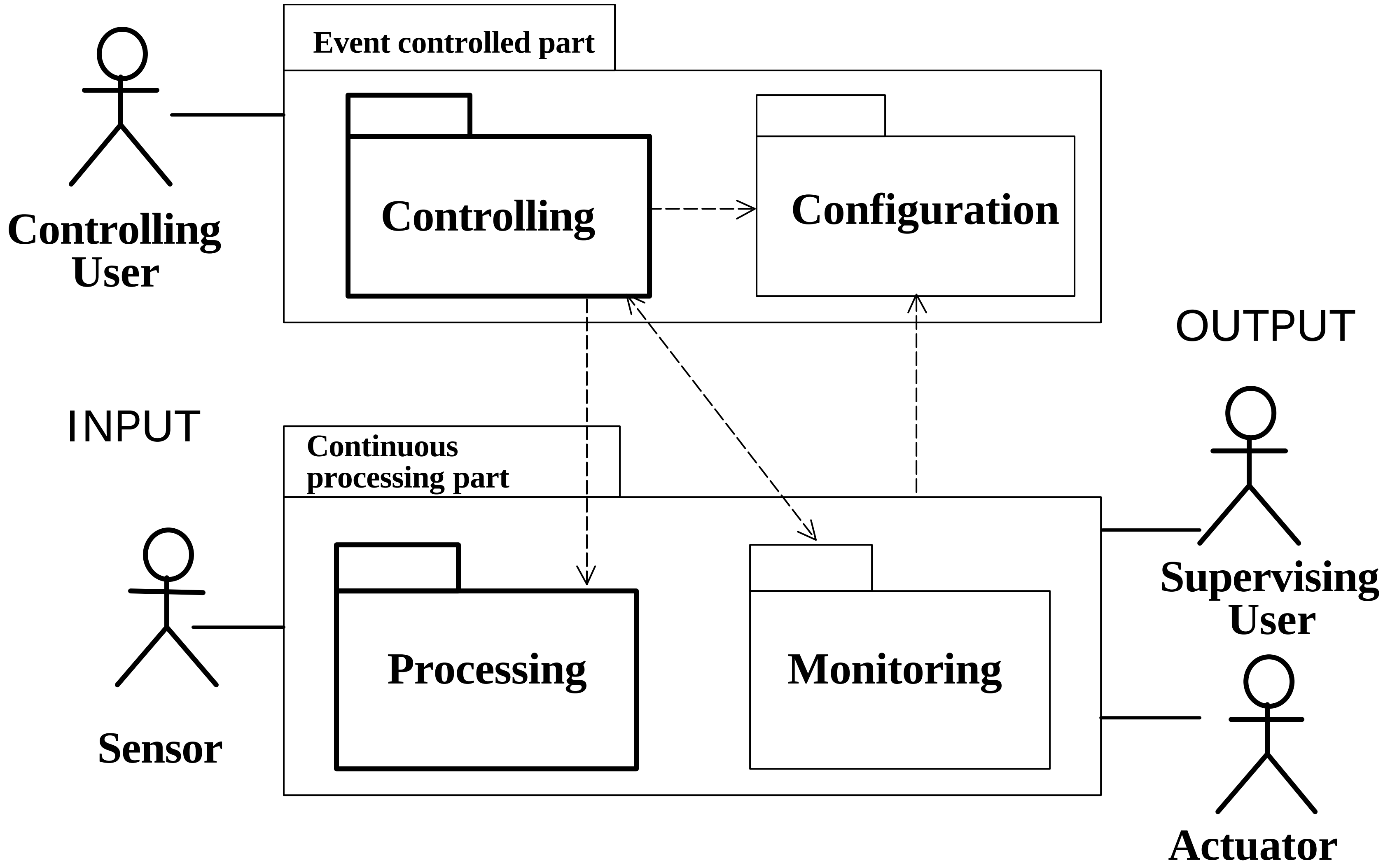
2. StartCommand::execute(MotorState *pS)
{ pS->start(); }

3. MotorStopped::start()
{ ....->SetActiveController(startController); }

Two types of VLT Use Cases



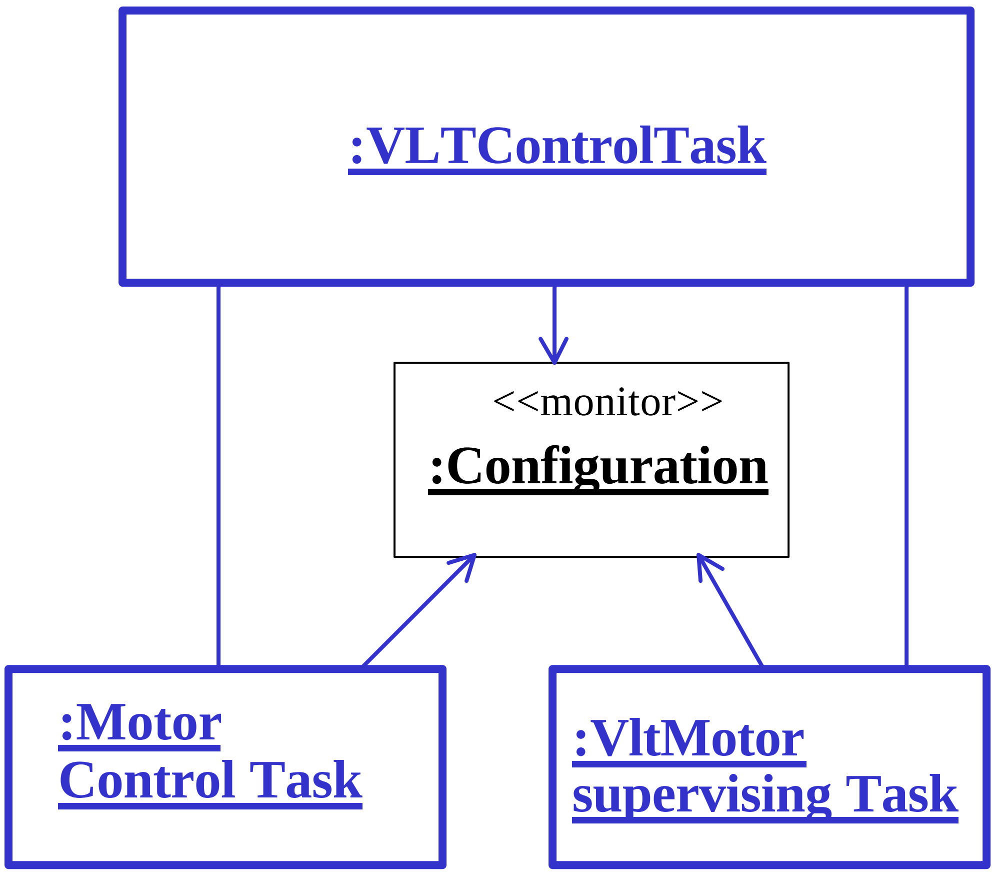
Generalised two-part architectural model



Task model example

Event
controlled part

Continuous
processing part



Experiences with Design Patterns

- Design patterns have been a very useful design tool
- The continuous part can be implemented with the *Strategy pattern* working in concert with the *Pipes and Filter pattern*
- The discrete part can be implemented with the *State pattern* working in concert with the *Command pattern*

Other OO experiences

- Extensive use of abstract classes and polymorph operations in the design
  - the continuous part is fast
  - easy to extend with extension based on subclasses
- Smaller state machines than in the previous SA/SD-RT (Ward&Mellor) based design

Conclusion

- Successful use of OO technology in an embedded system, where the use of design patterns has resulted in a flexible object model
- The two-part architectural model has been a valuable design tool - and is useful as a general design principle
- A framework has been build based on OO techniques (i.e. design patterns)

References

[Bushmann96]: *A System of Patterns: Patterns Oriented Software Architecture*

[COT]: *The Centre for Object Technology (COT)* (http://www.cit.dk/COT/)

[Gamma95]: *Design Patterns: Elements of Reusable Software*

[Jacobson92]: *Object-Oriented Software Engineering – A Use Case Driven Approach*

[Shaw95]: *Comparing Architectural Design Styles*

[Shaw&Garlan96]: *Software Architecture: Perspective of an Emerging Discipline*

[UML97]: *Unified Modelling Language (UML)*, www.omg.org