Simple Boiler

Version 1 / 02-07-2020

Table of Contents

1. Introduction

The purpose of the boiler application is to regulate the temperature of the water in a boiler.
The boiler hardware consists of a tank with water, a temperature sensor, a heating element and a cooling element.
Of course this can be solved easily without any objects and design patterns. See the *SimpleBoiler* application. The problem with this solution is that if something changes, e.g. the algorithm or the hardware, we have to rewrite most of our code.
The purpose of our solution is to enable changes in algorithms, hardware (sensors and actuators) and operating system environments without major impacts in our code. To achieve this we used object oriented programming techniques and design patterns.

2. The Three Domains

First we divided the problem is three domains namely the *Delivery*, *Regulator* and *Control* domains. See Figure 1.

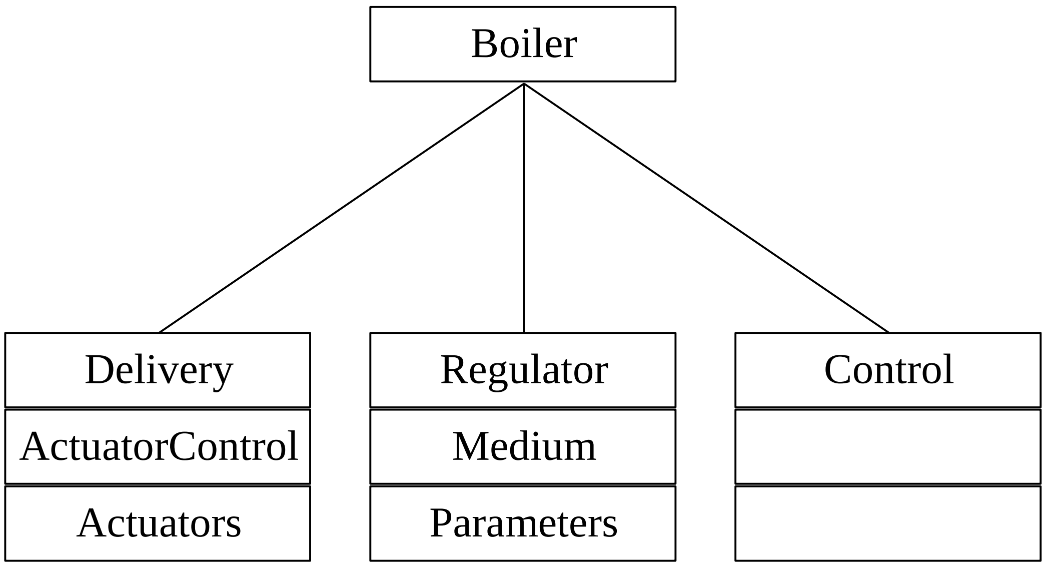

Figure 1: The Three Domains

The *Delivery Domain* encapsulates the resources of the system such as the actuators[1]. The *Regulator Domain* encapsulates the logic of the system and the parameters to regulate. The *Control Domain* takes care of the interaction between the system and the user.

Figure 2 shows the three domains as a UML class diagram. The *Boiler* is composed of the three domains.

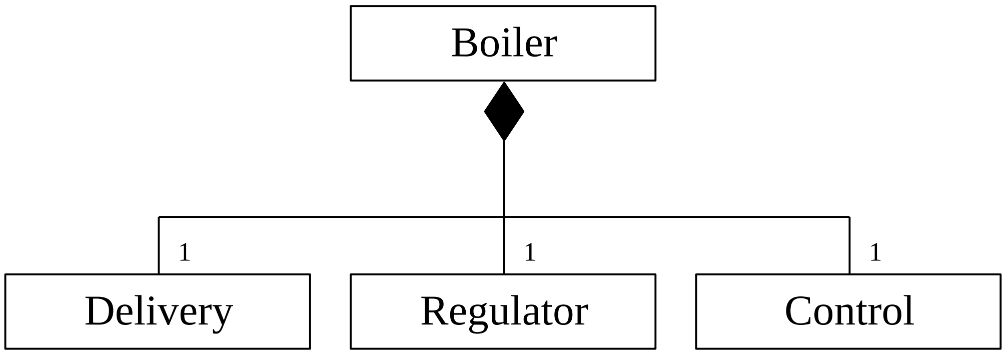

Figure 2: The Three Domains

All the domains talk to the *Boiler*. The *Boiler* thus is used as a **mediator**. A domain does not need to know the internal structure of the other domains. Hiding the internal structure for clients using higher abstraction objects is called the **facade pattern**.

[1] Actuators are used to enable/disable a resource such as a heater.

3. Events

The three sub domains communicate with each other using events. The three sub domains only know of the *Boiler* object. They send their message to the *Boiler* object and the *Boiler* passes the message to the correct sub domain. This is a kind of **message pattern**.
Every domain has its own *Event* base class that is derived from the *Event* class (see Figure 3).

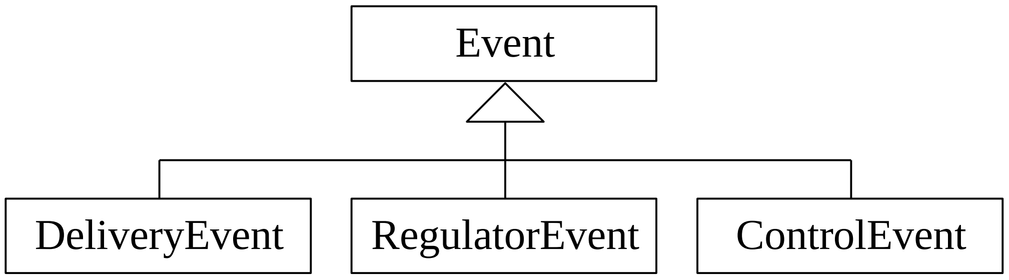

Figure 3: The Event base classes

The *Delivery Domain* has only one event (see Figure 4).

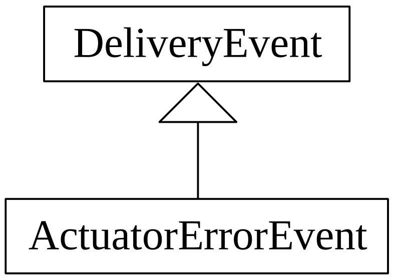

Figure 4: Delivery Events

The *ActuatorErrorEvent* will be sent when an actuator encounters an error. When an *ActuatorErrorEvent* is sent the *ShowActuatorError* method of the *Control* is called.

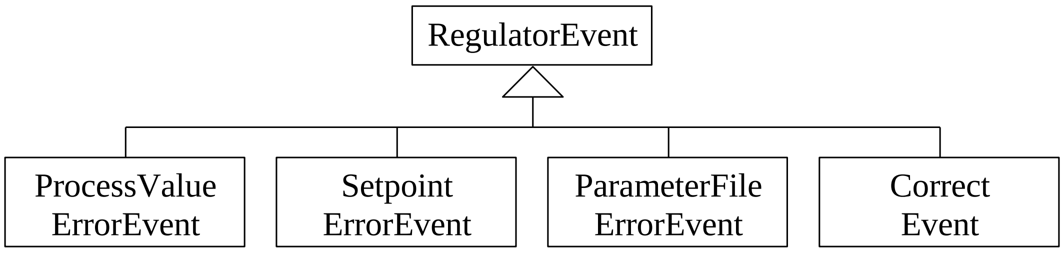

Figure 5: Regulator Events

The *Regulator Domain* has 4 events (see Figure 5). The *ProcessValueErrorEvent* will be sent when the process value of a parameter is out of bounds. The *SetpointErrorEvent* will be sent when the setpoint of a parameter is out of bounds. The *ParameterFileErrorEvent* will be sent when there was an error while reading or writing the parameter file. The *CorrectEvent* is send by the *Algorithm* to correct the output of the actuators.

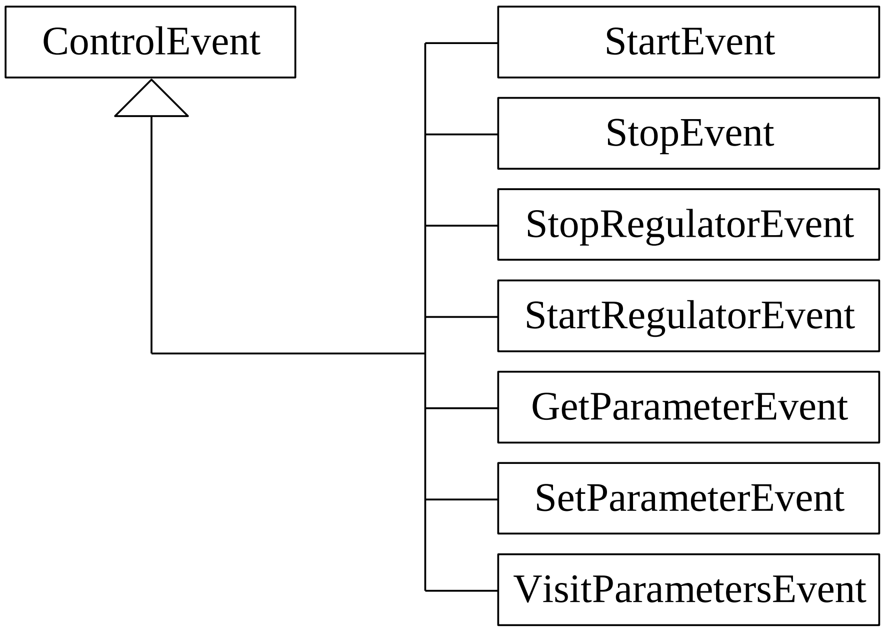

Figure 6: Control events

There are 7 events in the *Control Domain* (see Figure 6). The *StartEvent* and *StopEvent* are used to start and stop regulating a certain parameter. The *StartRegulatorEvent* and *StopRegualtorEvent* are used to start and stop the complete regulator. The *SetParameterEvent* and *GetParameterEvent* are used to set and get the parameter values like setpoint, process value and limits. The *VisitParametersEvent* is used to send all parameters to a **parameter visitor**. Using a **parameter visitor** we can in- or output all the parameter to a file or display device.

4. Factories

In the boiler application we used several factories (see Figure 7).

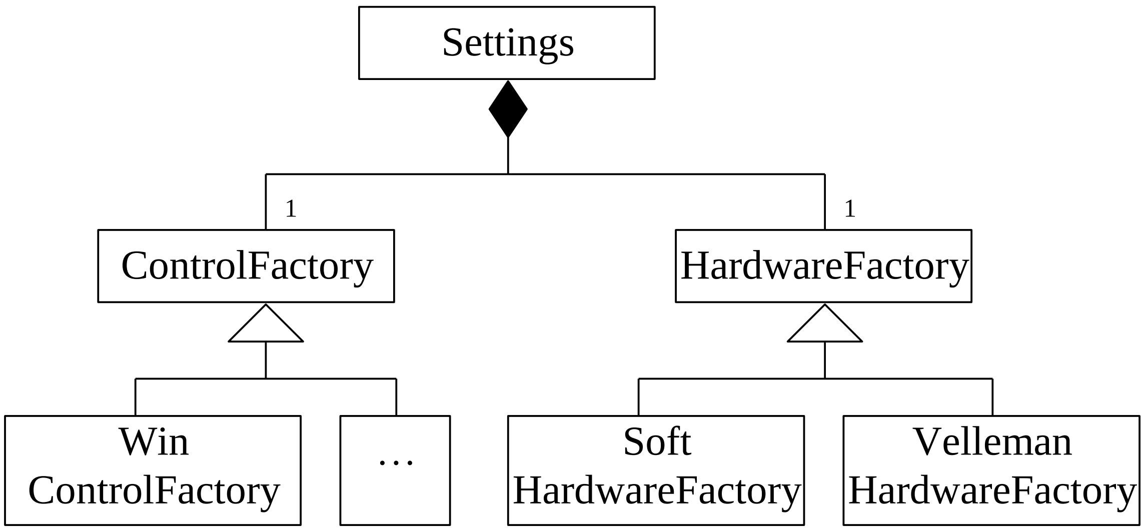

Figure 7: Factories

The *Settings* object is a globally available class that is queried by the system for several settings such as filenames and the creation of concrete objects like data store objects, actuators and sensors. The *Settings* object thus is a kind of **factory**.
If for example, there are more actuators available then only the concrete classes have to be made for them and the appropriate settings functions has to be adapted.

The *Settings* class uses two factory objects to create Operating System Environment and Hardware dependent objects. The *ControlFactory* is used to create a platform dependent *Control*. For now we only have a *WinControlFactory* but a *DOSControlFactory* or *UnixControlFactory* is possible.
The *HardwareFactory* is used to create hardware objects like sensors and actuators. The *SoftHardwareFactory* creates hardware objects that are emulated in software. The *VellemanHardwareFactor*y creates hardware objects that interface with the Velleman hardware.

5. Delivery Domain

The *Delivery* consists of an actuator control (see Figure 8). The *ActuatorControl* manages the available actuators. The *ConfigurationFile* is used to read the available actuators with their conversion strategies from a configuration file. The *ConfigurationFile* is implemented as a **singleton**.

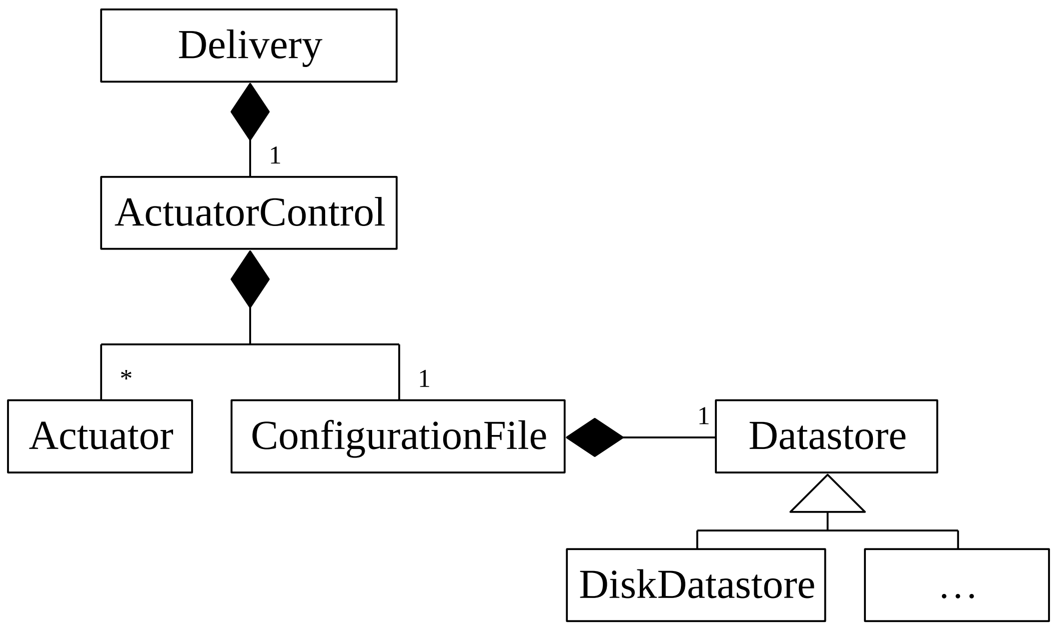

Figure 8: Delivery Domain

The *ConfigurationFile* uses a *Datastore* object that is generated by the *Settings* object. A concrete *Datastore* can for example be a disk or flash memory.
The *ConfigurationFile* uses the *Settings* class also for creating concrete actuators out of the actuator strings in the configuration file. Also the "Controller output to Actuator output" **strategy** is generated by the *Settings* object from the actuator string.
With the actuator control it is possible to have multiple actuators to regulate one parameter. Using the configuration file we can configure the available actuator and their response to the controller output at runtime.

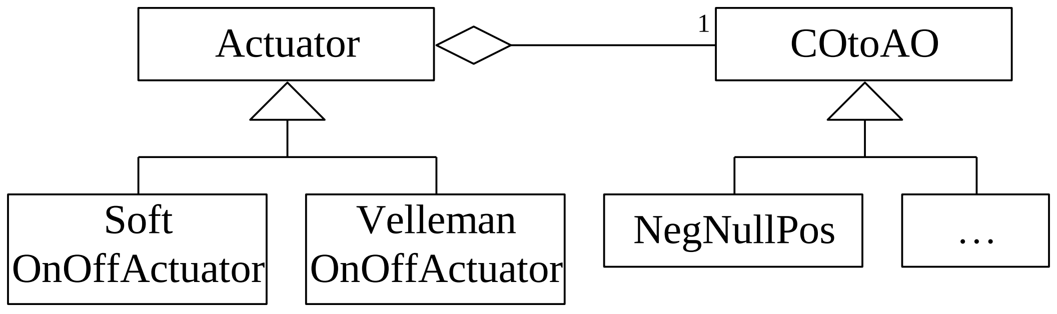

Figure 9: Actuators and 'Controller Output to Actuator Output' strategies

All the Actuators are derived from *Actuator* (see Figure 9). Currently we only have an *OnOffActuator*. The state of this actuator can only be on or off. Of course there are many other actuators possible for example actuators with analog outputs and actuators with error conditions.
Each actuator has a *COtoAO* **strategy** object to convert the controller output to the actuator output. Currently we only have a *NegNullPos* strategy. This strategy converts the controller output depending

on the sign of the controller output. This strategy is used to switch the *OnOffActuator* on or off. Of course there are many other strategies possible.
Since the actuators and their strategies are read from a configuration file we can determine the actuators and their behavior at runtime. If for example out cooler element breaks, we can remove the cooler from the configuration file and change the strategy of the heater to respond not so strong when the temperature reached the desired value.

6. Regulator Domain

The *Regulator Domain* consists of a *Medium* and an *Algorithm* (See Figure 10). The algorithm is implemented as a **strategy**. The *Algorithm* uses a **template method** to calculate the next controller output. The controller output is then send to the *Delivery*. The *Settings* class creates a concrete algorithm. Currently we have a simple algorithm that calculates the difference of the setpoint value and the process values. This difference is then send out as the controller output. Of course there are other algorithms possible like PID algorithms and fuzzy logic algorithms.

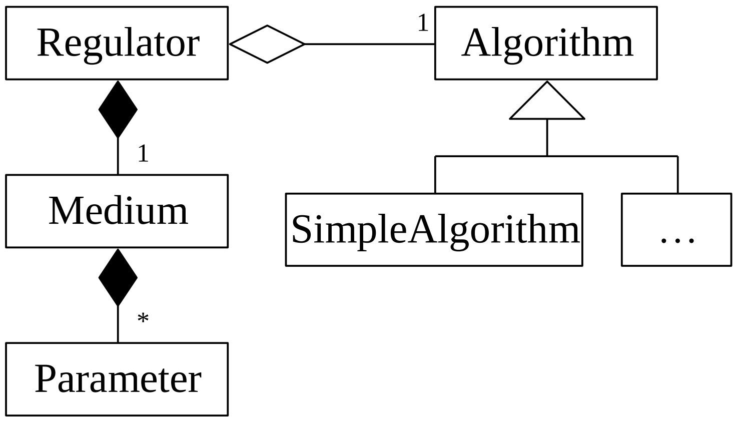

Figure 10: Regulator Domain

The *Medium* manages the available *Parameters*. The *Medium* provides functions to set- and get the properties of the parameter. The *Settings* class is used to create the available parameters so if we want extra parameters we do not have to change the *Regulator* or *Medium*. We only have to derive a new parameter class and update the *Settings* class.

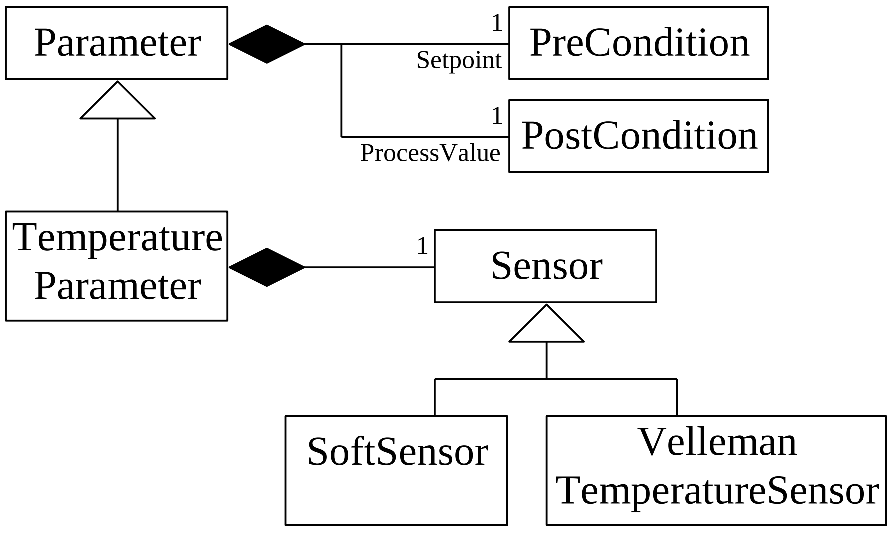

Figure 11: Parameters

Every parameter is derived from the *Parameter* class (See Figure 11). A parameter has a setpoint, process value, range, and a field to indicate if the parameter has to be regulated. The setpoint is a *PreCondition* property and the process value is a *PostCondition* property. For more information about the **property pattern** see the *Property Document*.
The *TemperatureParameter* has an associated *Sensor* object. The correct *TemperatureSensor* is created using the *Settings* class and the *HardwareFactory* object.

7. Parameter Visitors

A *ParameterVisitor* is used to in- or output the parameters from or to a device using the **visitor pattern**. Currently there are three concrete visitors (See Figure 12)

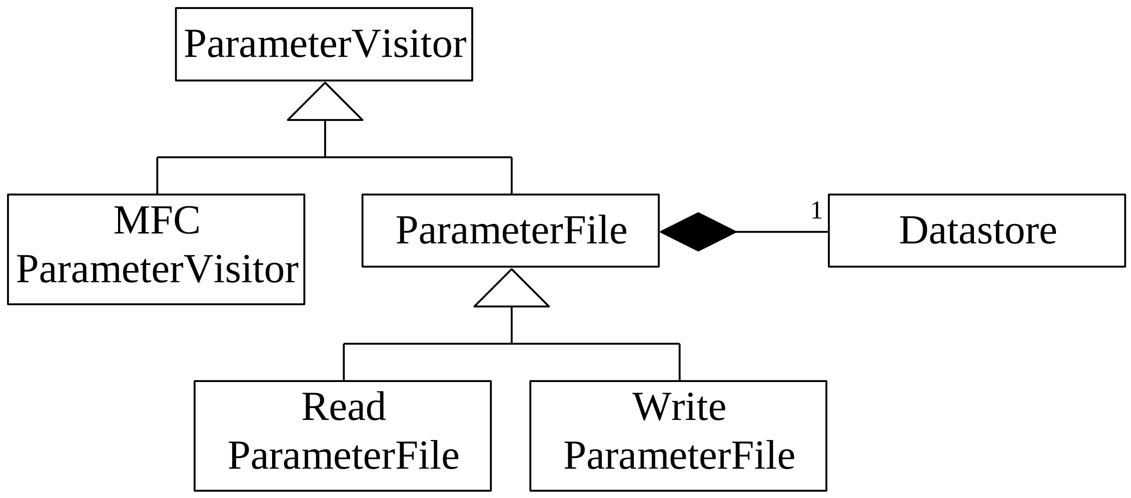

Figure 12: Parameter Visitors

The first two are the *ReadParameterFile* and the *WriteParameterFile*. These visitors are used to read and write the parameter properties (setpoint, range etc.) to a file. The actual reading and writing is done by a Datastore object thus the file storage medium can be a Disk or Flash memory etc. Whenever a parameter is updated the parameter file is written.
The *Medium* has a VisitParameters() method to send all parameters to a visitor. A *MFCParameterVisitor* is used by the *WinControl* to display the parameters in a list. More visitors can be created, for example to send the parameters to a log file, to send parameters to an Excel chart or to read and write parameters to the Windows registry.

8. Control Domain

The *Control Domain* consists of a concrete *Control* object (See Figure 13).

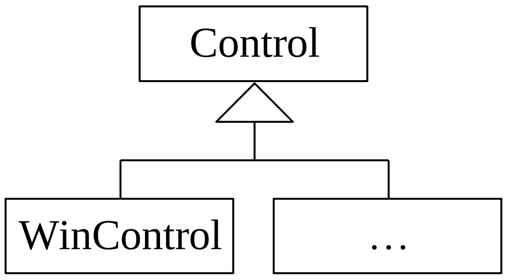

Figure 13: Control Domain

We have only made a *Windows Control* object that provides a user interface in Microsoft Windows. How the *Windows Control* is implemented is not interesting at the moment. Programming a Windows GUI is beyond the scope of this article. The interesting point is that, because a **factory** creates a concrete Control, we can easily provide other user interfaces without changing the rest of the program. We could for example create a DOS user interface.

9. Conclusion

Probably you noticed it already, except for the introduction, we rarely mention that this application is to regulate a *Boiler.* This means that the rest of the document is not depended on a *Boiler*. And this is exactly what we wanted. If we rename the *Boiler* class to for example *GenericRegulator* we have created an application frame that can be used to regulate everything, from the temperature in a boiler to the liquid level, flow, temperature and oxygen levels in a biological process. This all by only writing new specialized classes and adapting the factories.

10. Used Patterns