A FLEXIBLE AND TESTABLE SOFTWARE ARCHITECTURE: APPLYING PRESENTER FIRST TO A DEVICE SERVER FOR THE DOOCS ACCELERATOR CONTROL SYSTEM OF THE EUROPEAN XFEL

A. Beckmann, S. Karabekyan, J. Pflüger, European XFEL, Hamburg, Germany

Abstract

Presenter First (PF) uses a variant of Model View Presenter design pattern to add implementation flexibility and to improve testability of complex event-driven applications. It has been introduced in the context of GUI applications, but can easily be adapted to server applications. This paper describes how Presenter First methodology is used to develop a device server for the Programmable Logic Controls (PLC) of the European XFEL undulator systems, which are Windows PCs running PLC software from Beckhoff. The server implements a ZeroMQ message interface to the PLC allowing the DOOCS accelerator control system of the European XFEL to exchange data with the PLC by sending messages over the network. Our challenge is to develop a well-tested device server with a flexible architecture that allows integrating the server into other accelerator control systems like EPICS.

TECHNICAL BACKGROUND

The European X-Ray Free-Electron Laser (XFEL) Facility will use three ~200m long undulator systems to produce laser light by the Self Amplified Spontaneous Emission (SASE) process [1]. The photon energy can be varied by a change of the undulator gap. The corresponding motion control is implemented with Beckhoff TwinCAT PLC software on Windows PCs.

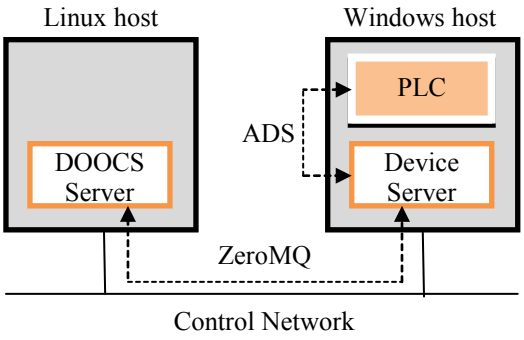

Figure 1: Integration of undulator control into DOOCS.

Figure 1 shows how undulator control is integrated into the Distributed Object-Oriented Control System (DOOCS), which is the accelerator control system of the European XFEL facility [2]. The DOOCS Server, running on a Linux host, defines for the undulator system a set of properties, such as the undulator gap. These properties can be read or modified by DOOCS client applications (not shown in the figure) in order to control the undulator system. The DOOCS Server exchanges the property values with the Device Server using the ZeroMQ message transport library [3]. The Device Server runs on a Windows host together with the PLC of the undulator system. Both exchange the property values using the Beckhoff Automation Device Specification (ADS) protocol.

The reason for having two servers is that ADS is supported only for Windows platforms and the DOOCS Server is supported only for Linux platforms. The message interface between both servers crosses this platform border. An additional benefit of the message interface is that it adds flexibility on the machine control side. Anything may connect to the Device Server, regardless of the platform it runs on, as long as it is able to generate appropriate messages. It could for example be an EPICS Channel Access Server (CAS) with only little effort to implement the message interface.

DESCRIPTION OF PRESENTER FIRST

So, how does Presenter First help in developing the Device Server application? Presenter First proposes a pattern for structuring the code in a specific way, and a process for developing the application in a specific sequence [4]. The pattern improves testability of the code and offers some flexibility regarding the interaction with the environment. The process saves effort by developing on the basis of the intended behaviour of the application.

The Pattern

Applications are implemented using a variant of the Model View Presenter (MVP) pattern, as shown in Fig. 2.

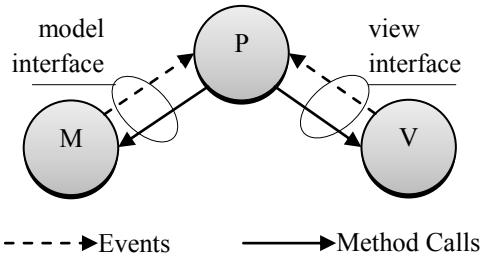

Figure 2: MVP variant used in PF.

The presenter represents the behaviour of the application that is defined by the functional requirements, the model manages the application data and logic, and the view interacts with the environment. Model and view do not communicate directly with each other because this would limit flexibility; instead both communicate with the presenter over clearly defined interfaces. They send events to the presenter to trigger some behaviour, which

usually results in method calls back to either model or view.

The use of interfaces is essential for the testability, because it decouples the presenter from the model and view. The presenter depends only on the interfaces and not on the objects that implement the interfaces. Within the test environment the model and view can then be replaced with a mock model and mock view that provide the same interface but behave as required by the test.

Flexibility comes with the separation of code, which interacts with the environment, into the view. This part can be changed without the need to modify either model or presenter. It just has to implement the view interface. Currently, the device server uses ZeroMQ to interact with the network. If this needs to be changed, only the view requires modifications reducing the effort to do so.

The Process

The development of an application starts with the presenter. The functional requirements, which define the application behaviour, are analysed for their impact on the model and view. The result is the definition of events and methods for the model and the view interfaces. Once all requirements have been analysed, the interfaces are completely defined and implementation of model and view may now start.

The advantage of this approach is that the application exactly fulfils the functional requirements: it behaves exactly as intended. Over-engineering by implementing functionality that is not required is avoided, which saves development effort and reduces the complexity of the code.

DEVICE SERVER DEVELOPMENT

The use of Presenter First is described by taking the Device Server as an example. The Device Server is a C# application that runs as a Windows service on the host with the PLC. Unlike regular applications, a Windows service is started by the Windows Service Control Manager (SCM) during boot and stopped during shutdown.

Software Architecture

Figure 3 shows the basic software architecture of the Device Server. It is divided into two large modules: the Windows Service module and the Control module.

Windows Service contains code that registers the Device Server with the SCM. Its only purpose is to start and stop the Control module whenever the Device Server itself is started or stopped by SCM.

Control contains the instances of model, view and presenter, which together implement the applications functionality. Once the Control module is started by the Windows Service module, the presenter processes events from either model or view according to the implemented behaviour. Events from the model are triggered by the PLC, for example when PLC variable values have been changed. Events from the view are triggered by the ZeroMQ interface for example when a message arrives.

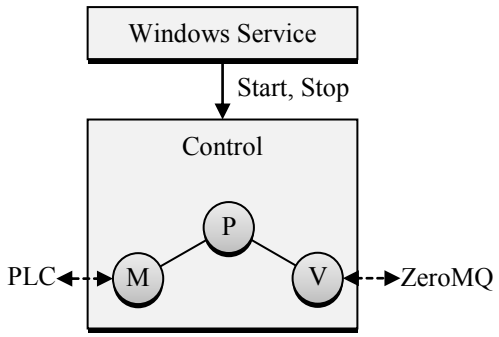

Figure 3: Software architecture of Device Server.

Applying Presenter First

As proposed by Presenter First the development starts with the presenter by analysing the functional requirements of the Device Server. These are:

1. The service shall activate or deactivate the message interface, when it is started or stopped by SCM respectively.
2. The service shall respond to request messages received at the message interface.
3. The service shall send an update message via message interface, whenever values of PLC variables change.

In the following, the analysis is described taking the second requirement as an example. It starts with a request message arriving at the ZeroMQ message interface. A message is in principle a byte array with a certain length. ZeroMQ does not care about the content; it just tells the view that a number of bytes have been received. The parsing of the message content is considered as being part of the application logic, so that only the model knows how to do it. Therefore the incoming message needs to be passed to the model. To do so, the view sends an event to the presenter that a request message has been received. The event handler inside the presenter then reads the message bytes from the view, translates them into a more convenient to use message object, and passes it to the model. The model processes the message object and returns another message object as a response. The presenter translates the response message object back into a byte array and passes it to the view. Finally, the view sends a response message on the ZeroMQ message interface.

The analysis results in the following interfaces. The view interface requires:

- an event indicating that a message has been received,
- methods returning the length and bytes of the received message,
- a method to set the length and bytes of the response message

The model interface requires:

- a method to process the received message, returning a response message

The remaining task is now to implement the events and methods from these interfaces inside the model and view.

Testing

The Device Server is tested on unit and on system level. Unit level tests verify the functionality of individual software units, such as the presenter. System level tests verify the integration of the complete server into the environment, which consists of the Windows SCM, the DOOCS Server and the PLC.

Some parts of the Device Server cannot be tested on unit level, since they interact with the environment that cannot be generated during an automated unit test run. These are:

- the Windows Service module
- the view
- the part of the model that communicates with the PLC via TwinCAT ADS Communication Library

To reduce the probability of errors in these parts, their code is as simple as possible, which means that there is almost no processing inside these parts. This also reduces the effort to test these parts manually on system level. The Windows Service module is very simple since it just passes down start and stop from the SCM to the Control module. The view is a little bit more complex since it contains some inevitable processing related to the use of ZeroMQ. Finally, the ADS library, which is used by the model to communicate with the PLC, is wrapped with an ADS interface adapter. Instead of calling methods directly from the ADS library, the model calls methods from the interface adapter. Much like the interface between presenter and model, this decouples the model from the ADS library and increases its testability. The adapter itself is very simple, since the interface methods simply call the corresponding ADS library methods.

The presenter is tested on unit level by replacing model and view with mocks that implement the same model and view interfaces. Mocks are specialized test case objects, which in principle track calls to the interface methods. At the end of a test, the mocks verify that the methods have been called the correct number of times. They may also verify that the correct arguments have been passed, for example that a method has been called once with an integer argument equal to 1. For more complex test cases, mocks can be set up to return values, or even to raise program exceptions. Tracking method calls with mocks is known as behavioural verification, in contrast to the classical state verification, where data values (or state of data) are checked after some part of the code has been executed [5]. It is therefore ideally suited to complement Presenter First with its focus on the behaviour of an application.

A typical presenter test case creates instances of the presenter, the mock model and the mock view. The mocks themselves are taken from the mock object library Moq [6]. The test case then configures the mocks and starts the test by triggering one of the events from either the model or view interface. At the end of the test case the mocks perform their verification by checking whether methods have been called the correct number of times and with the correct arguments.

The model is tested on unit level much like the presenter by replacing the ADS interface adapter with a mock adapter. However, test cases start with a call of a method from the model interface instead of triggering events. There is a test case for each model interface method, so that the model is almost completely tested except for the ADS interface adapter at the boundary that communicates with the PLC, as mentioned earlier.

Finally, the Device Server is tested on system level. It is installed as a Windows service on the host with the PLC. Then, the DOOCS Server and PLC are used to stimulate the Device Server. Afterwards the state of the DOOCS Server and PLC are verified manually.

Experience

The use of Presenter First allows testing most of the functionality already on unit level. This notably reduces the probability of errors detected on system level. The code is cleanly structured, which simplifies debugging and refactoring. The developped Device Server runs stably over a longer period in a test setup with two undulators controlled by one DOOCS server.

SUMMARY

Presenter First has been used to develop a well-tested Device Server for the undulator control system of the European XFEL facility. The software architecture provides flexibility regarding integration into the machine control system.

The Presenter First software architecture increases the testability of applications with event-based behaviour, which is important for mission critical software such as the undulator motion control Device Server. Development is guided by the behavioural requirements and results in compact code that does exactly what is required and nothing else.

Flexibility comes with the separation of interface and application logic. The interface can be changed without the need to adapt the application logic.

REFERENCES

[1] M. Altarelli et al. (ed), “The European X-ray free-electron laser Technical Design Report”, DESY Report 2006-097, 2006

[2] K. Rehlich et al., “The Accelerator Control Systems at DESY”, in ICFA Beam Dynamics Newsletter No. 47, pp. 139-166, 2008

[3] http://www.zeromq.org/

[4] M. Alles, D. Crosby, C. Erickson, B. Harleton, M. Marsiglia, G. Pattison, C. Stienstra, “Presenter First: Organizing Complex GUI Applications for Test-Driven Development,” Agile 2006, Minneapolis, July 2006

[5] http://martinfowler.com/articles/mocksArentStubs.html

[6] http://code.google.com/p/moq/