ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO2BC004

APPLYING STANDARDISED SOFTWARE ARCHITECTURAL CONCEPTS TO DESIGN ROBUST AND ADAPTABLE PLC SOLUTIONS *

S. T. Huynh†, B. Baranasic, M. Bueno, T. Freyermuth, P. Gessler, N. Jardón Bueno, N. Mashayekh, J. Tolkiehn, L. Zanellatto, European X-Ray Free-Electron Laser, Schenefeld, Germany

Abstract

Between evolving requirements, additional feature requests and urgent maintenance tasks, the Programmable Logic Controllers (PLC) at the European X-Ray Free Electron Laser Facility (EuXFEL) have become subjected to an array of demands. As the maintainability effort towards the existing systems peak, the requirement for a sustainable solution become an ever pressing concern. Ultimately, in order to provide a PLC code base which can easily be supported and adapted to, a reworking was required from the ground up in the form of a new suite of libraries and tools. Through this, it was possible to bring standardised software principles into PLC design and development, conjunctively offering an interface into the existing code base for ongoing support of legacy code.

The set of libraries are developed by incorporating software engineering principles and design patterns in test driven development within a layered architecture. In defining clear interfaces across all the architectural layers - from hardware, to the software representation of hardware, and clusters of software devices, the complexity of PLC development decreases down into modular blocks of unit tested code. Regular tasks such as the addition of features, modifications or process control can easily be performed due to the adaptability, flexibility and modularity of the core PLC code base.

INTRODUCTION

Over the decades, Programmable Logic Controllers (PLC) have been programmed and the world in which it exists, has also evolved into the standard that is known of today as the IEC 61131-3 [1] standard. With the increase in memory and the greater integration of Structured Text (ST) [2] PLCs have started to shift from an electrical hardwired concept to all that is offered within a software programming language. This cross discipline has opened up a powerful feature set which is often underutilised, and provides PLCs with the opportunity to integrate core software principles into their code base.

In the process of redeveloping the PLC code at the European X-Ray Free Electron Laser Facility (EuXFEL), the PLC developers are committed to developing a code base which is designed with thought and care, bringing into the design many of the software practices which are often embodied in large software projects, whilst taking into consideration the needs and functions of the PLC as the facility expands and refines its needs. Taken into consideration is the continual support and maintenance of the current legacy code base, and its future integration.

TECHNICAL DEBT

When juggling between time constraints and numerous feature requests, it would not be uncommon for code to be added to an existing code base in a haphazard manner, without thorough testing. Whilst this approach can seem to work, the consequences will inevitably catch up with the developers.

This was precisely the situation that the PLC developers at EuXFEL found themselves in: more time was being dedicated towards maintenance and resolving existing issues within the existing code base, than developing additional functionality. Within a scientific environment where experiments are in a constant state of flux, the demands on the PLC to be agile and adaptable, but also reliable was also further stressed.

It became customary to develop new patches to get around limitations, and the task of maintaining the array of existing tools became unmanageable. This is especially noted where core libraries and functions that were heavily relied upon within various programming languages, became deprecated or superseded.

Due to the amassed technical debt which accumulated over several years, the tipping point for redevelopment was reached. As such, a complete redevelopment was required for the PLC code base.

KEY REQUIREMENTS

As with any major software project, the first step was the development of a Software Requirements Specification (SRS). As this paper will focus on the non-functional design aspects, the key outcomes of the SRS which ultimately defined the final PLC design, are detailed below.

Architectural Decision Records

One of the challenges in maintaining legacy code is understanding what historical decisions were made and their reasoning. Unfortunately, this is often neither obvious nor something that can be garnered with going through the existing code. To be able to refactor, extend or at times, remove redundant code can be challenging without this information.

Knowing this will allow one to refine or edit existing work with an awareness of the original intention. Without this, code modification can result in unintended consequences, especially in a code base without a clean architecture. This

* Work supported by ...
† sylvia.huynh@xfel.eu

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO2BC004

can easily be overcome with up-to-date documentation of all Architectural Decision Records (ADRs) [3].

ADRs not only record the final decision, but more importantly, the justification, options considered and known constraints and any assumptions that were made during decision making. A common ADR outline can be found below:

- **Topic:** The design aspect which is being addressed. Eg.'Array Bounds Declaration'
- **Issue description:** A detailed explanation of why this issue exists and where it would be used/implemented
- **Decision:** The final and succinct decision that was made. This should not be open to interpretation.
- **Status:** Current status of the decision. The following are used at the EuXFEL: Proposed, Accepted, Rejected, Superseded, Deprecated.
- **Assumptions:** The underlying assumptions which the decision is being made upon. This can be related to the decision-cost, constraints, technology etc. These environmental factors can impact the decision, and under a different set of circumstances, may result in a different decision being made.
- **Constraints:** Additional constraints that are imposed upon the issue
- **Positions:** A list of positions which were considered during the decision making process. This should capture a list of alternatives, why they were ruled out, what advantages they may have offered or subsequent issues they added.
- **Justification:** The reason that a particular position was selected. This is often equally, if not more important that the decision itself, as it will help others understand the rationalisation of the decision.
- **Implications:** All decisions have implications. This should outline the foreseeable implications of the decisions and indicates how well thought-out a decision was made.
- **Related resources or decisions:** Decisions are often not made in isolation, and can impact previous decisions made, or put into action future decisions. Any associated resources and ADRs can be listed here to aid referencing.

This enables all developers current and future, to understand why the code was developed in a particular way, and how the decision came to be, eliminating any unnecessary guesswork. When appropriate, ADRs can also act in place of discussion minutes, and product documentation.

Layered Architecture Pattern

Each layer within a layered architecture pattern [4] has a distinct and well defined function. This ensures that changes within one layer will not impact another. This provides a clear boundary of scope for the responsibilities within each layer, making it easier to adapt and scale. It can also aid development by limiting the problem at hand.

When we think of passing signals from hardware to software, we naturally consider processing the signal in steps. For example: the hardware raw units, the corresponding PLC bits, to the scientific/user unit, to a meaningful value within a component, and finally, to a value which can be interpreted by the Supervisory Control and Data Acquisition (SCADA) system. This stepped approach as seen in Fig. 1 naturally adapts itself to a multi-layered pattern within the software architecture. Many common software systems such as UNIX or any communication protocol also implement such a architecture.

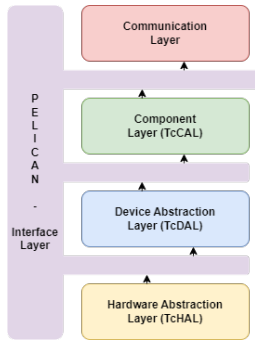

Figure 1: PLC architectural layers.

A key benefit of a layered approach lies within code entanglement. As your code base grows, it naturally becomes more intertwined. With a layered architecture pattern, the boundary of each layer will intrinsically limit this. The layers of the PLC Library will be further elaborated upon and explored within the **PLC Architecture Overview** section.

Interface Concept

To ensure each layer within the code base can continue to interact with each other while being heavily refactored or edited, an interface is required.

An interface defines the expected data structure, type and value from which all information is to be formatted before it can pass between the various layers. The concept of interfaces [5] is one of the most critical in software engineering. It ensures that the various code functions, classes and modules can be interacted with in an expected manner. It allows the user to know what data they can access, whilst the technical details regarding the implementation is hidden.

An example of an interface for a voltage derived signal can be seen in Fig. 2. Here, it is possible to see the list of available features which can be accessed by all other parts of the code base.

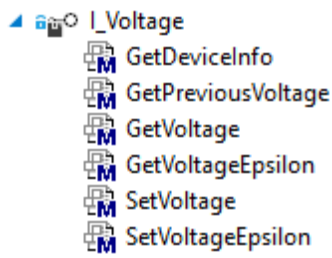

Figure 2: Voltage interface.

In having clear and well defined interfaces, the function of

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO2BC004

the interface can be completely independent to how the piece of information is used. The interface acts as a guarantee to the accessing method and implementer, of what can be expected and what is to be delivered. It can be treated as a contract between provider and consumer of information, which allows both parties to program against this contract individually without knowing any implementation details of the other side.

Modularity

As PLCs are used to integrate in hardware devices for remote control, the high repetition of code leads to the requirement for modularity. For example, if a piece of hardware was to be integrated into the PLC, the device will have similar properties such as an error, a predominate value of note, and configurable aspects; not unlike other hardware devices. The error handling, configuration methods or value scaling can all be treated as individual modules, which can then easily be replicated across all the various devices.

Modular code ensures that each code block is limited to providing a single function. This results in smaller code blocks, which in turn make it easier to read, diagnose and to also reuse. Within the TwinCAT environment, these modules are implemented as objects or Program Organization Unit (POU). As each POU or object will have a similar interface, take for example, Error Handling; the implementation will then be encapsulated and specific to each hardware device as required. In having smaller code blocks dedicated to a single feature, the code also makes itself easier to test and reason about.

Test Driven Development

A novel concept referred to as Test Driven Development (TDD) [6] within software development, outlines that code should only be written to ensure a previously written test succeeds. Whilst this can be seen as being counter-intuitive, this ensures that a method and interface has been well thought out before any production coding has even begun. Only once a test has been developed can any code be written. The code base is now being driven by failing test cases, rather than tests being developed to reflect the written code. The TDD process can be seen in more detail in Fig. 3 with corresponding tests as seen in Fig. 4.

A key advantage to this approach is the knowledge that comes with a code base with a high test coverage, where any function or method will have an existing test case. Especially so as code coverage can not be measured within TwinCAT. This provides some level of security and reliability when all tests succeed during a major refactoring.

TcUnit [7] provides a library which can easily be integrated into the TwinCAT environment. Within each library developed, a program is created specifically to run the unit tests, ensuring a functional and error free code base. Another major benefit of using TcUnit lies in its ability to be added into a continuous integration/continuous development and deployment (CI/CD) pipeline, further enhancing code quality and reliability.

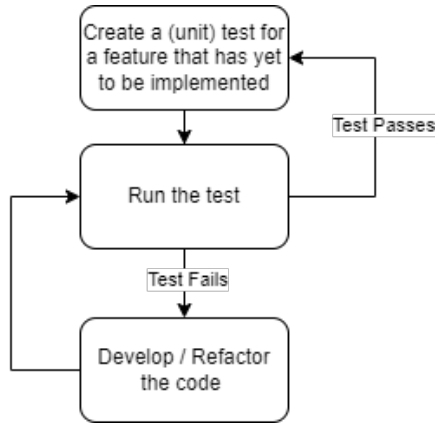

Figure 3: Test Driven Development process.

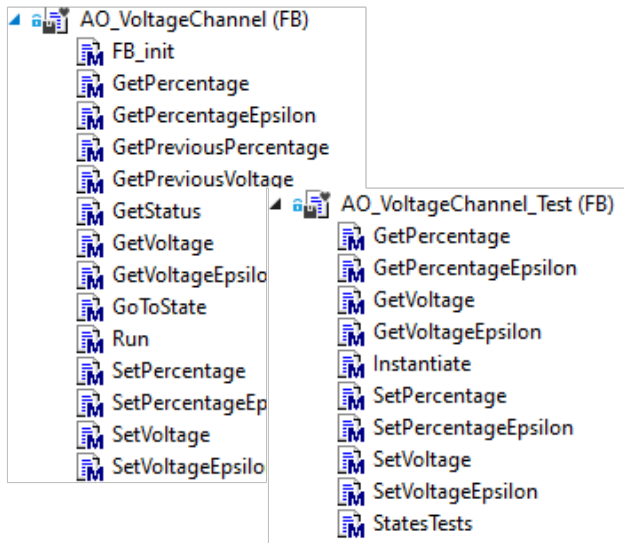

Figure 4: Test Driven Development example.

Support of Legacy Code

Given the amount of work which has already gone into developing, testing and commissioning of the various hardware components within existing code base, it would be counter productive to completely disregard what has been done thus far. In an attempt to salvage this effort, one of the key requirements would be to provide a way to adapt the legacy code into the new architecture.

Whilst a lot of the design discussion revolved around what can be done to rectify some of the key restrictions or how it can be improved upon, a fair amount of effort was put into ensuring that it was possible to provide ongoing and continual support for the legacy code base within the new architecture.

Due to a layered architecture pattern, the integration of the legacy support can constitute its own module along side the device abstraction layer in a straight forward and simple manner. It only requires a relatively simple adapter to replace the old direct hardware linking to adopt old devices to the new TcHAL.

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO2BC004

PLC DEVELOPER: AN APPROACH

Software principals and patterns are often ideal, and are wildly utilised within major software projects. An overview of the current PLC code base at the EuXFEL is detailed below, along with the application and implementation of the software design patterns and concepts previously mentioned. Supporting software design concepts which provide a backbone to many of the software objects are also detailed.

PLC Architecture Overview

The PLC code base comprises of a collection of libraries, which together form the TcZookeeper suite [8], along side with some tools integrated into TwinCAT (VisualStudio) to assist using these libraries. An overview of the TcZookeeper suite can be seen in Fig. 1.

A single architectural layer can be made up of one or several libraries, where each library will be responsible for a single domain with a clear separation of concern. It was decided to keep each architectural layer as an independent library or set of libraries to enforce a clear boundary, but also to aid deployment. As PLC projects are generated or created, they will reference the necessary TcZookeeper libraries. To minimize the requirement of PLC updates, the libraries are all abstracted according to their functional layer and only updated within a PLC project when required. Additionally, a library can also be easily added or swapped out, without impacting an existing references. For example, a library on the communication layer may be responsible for providing the OPC Unified Architecture (OPC UA) protocol, and another library also within the communication layer which is catered to a different protocol. Both can exist simultaneously, or independently as defined by the connecting SCADA system.

TcHAL The lowest layer within the TcZookeeper as seen in Fig. 5 is the TwinCAT Hardware Abstraction Layer (TcHAL). The fieldbus hardware configuration is performed at this level, and each fieldbus device or EtherCAT terminal is encapsulated within its own POU. Data read from the EtherCAT terminal and devices are converted into meaningful International System of Units (SI) where possible.

TcDAL The TwinCAT Device Abstraction Layer (TcDAL) is the representation of a piece of hardware that is to be controlled and integrated within TwinCAT as seen in Fig. 6. It is aimed for this layer to represent basic devices such as valves, pumps, pressure gauges, motors and the like.

Essentially, the functionality and how the various device signals are to be interacted with, are all encapsulated within the various objects or POUs of this library. This detaches itself entirely from the type of signal itself. Take for example, a pressure gauge. The gauge value consists of just an analogue value in mbar. At this level, it is irrelevant if the value was originally derived from a voltage or current source, that should have been handled within the TcHAL.

TcCAL As the facility consists of multiple beamlines, there will be repetition around some beamline components, such as a vacuum section as seen in Fig. 7. A vacuum section will always constitutes of the same key parts - a collection of pumps, pressure gauges and valves. The functioning of this component on a higher level will also be similar - A set of valves and pumps will close, open, turn on or off to maintain a predefined pressure during a venting or evacuating process.

The logic and control process for a more complex device which is comprised of multiple low level devices taken from the preceding TcDAL layer, is defined within this library.

Communication This library deals with the communication protocol in order to pass data between the PLC and the SCADA system. By isolating the communication protocol into a single layer as seen in Fig. 8, the potential to easily interchange or apply multiple communication protocols simultaneously becomes feasible.

This ensures that the behavioural aspects of a hardware device integration are entirely encapsulated from all of the code dealing with the communication. Whilst this may seem obvious, this was a key bottleneck in the previous iterations of the PLC framework at the EuXFEL.

Pelican The Pelican is the library which is responsible for all of the interface definitions and the interface manager. This layer encapsulates all the available interfaces within a single libraries across the entire TcZookeeper. For any layer to interact or pass information to another layer, it must use one of the available interfaces provided by the Pelican.

Finite State Machines

Data Structure In order to keep the code as modular as possible, an object will contain device information within its interface, which is created by a data structure with the properties as shown in Fig. 9. This will be applied to every object, whether it be a valve or a fieldbus terminal etc.

Every device will contain the *Configuration*, *Error* and *Operating* states, in addition to any device specific states as required which is known as its *Process* state.

FSM Implementation Each device will then implement a Finite State Machines (FSM). Following a version of the [9] state pattern with some adaptations to PLC specific constraints. An example of this can be seen in Fig. 10.

The main trade off comes down to complexity versus consistency. Simple devices will be taxed with a high code overhead, while complex devices will benefit in clarity when implementing the FSM following the state pattern. The major benefit however, lies in the standardisation and modularity of all devices and objects.

Peer Communication In turn, this simplifies the concept for peer communication, or inter device interaction. If a component was developed within TcCAL, a controlling device can easily and consistently obtain the running state of all its secondary devices.

If a device requires interlocking commands, this can also be implemented on the PLC project level rather than within

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO2BC004

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

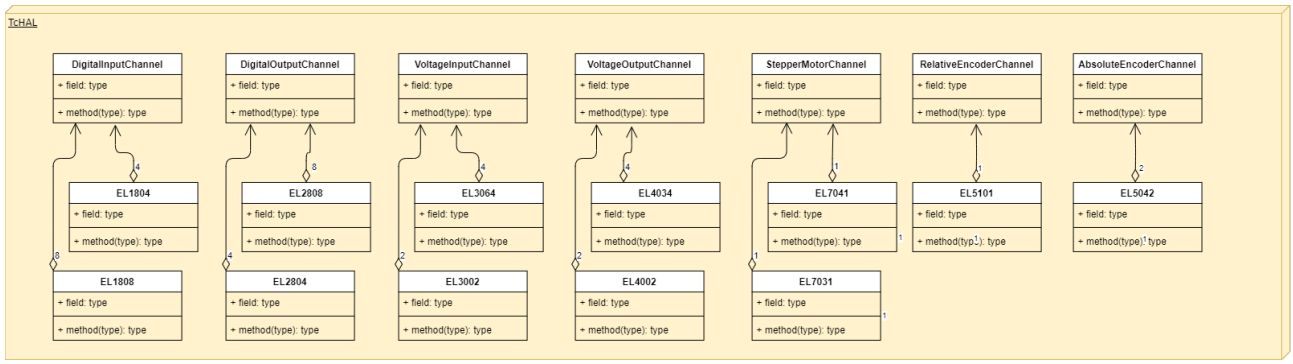

Figure 5: TcHAL example.

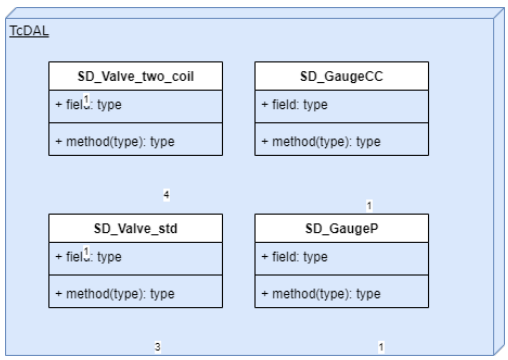

Figure 6: TcDAL example.

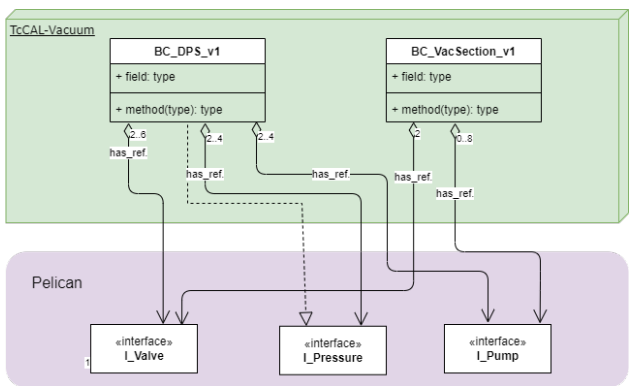

Figure 7: TcCAL example.

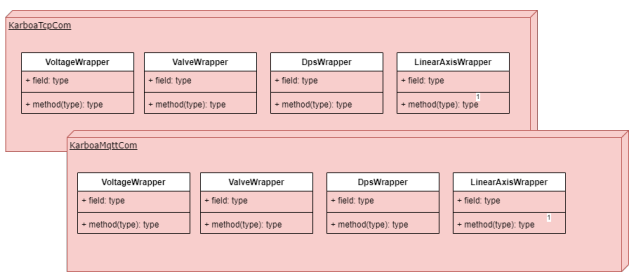

Figure 8: Communication layer example.

the library, using the exact same methodology via the use of the standardised FSM interface.

This enables various developers to be able to quickly understand the functioning of every device, without requiring all of the specialised in-depth knowledge of a PLC device integrator. The core functionality is encapsulated away within the state implementation. This dividing of knowledge makes it easier for everyone to interact with at the level they need or are most comfortable with.

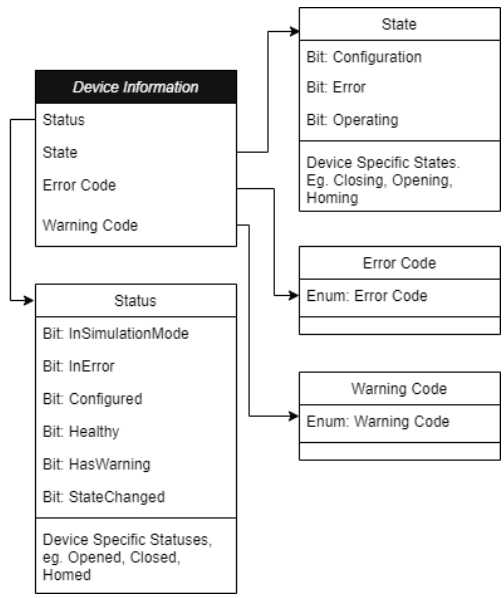

Figure 9: Device information.

Observer Design Pattern

To aid much of the interaction between the various layers, an observer pattern was also implemented. With all of the interfaces available, an observer pattern can assist in keeping track of which objects are being managed or depended on.

The observer pattern [9, 10] details how to design a way to pass information between objects with dependencies, in a consistent manner. An object can request to register itself to another object as an observer, thus subscribing to updates from the object which is being watched. Once the observing object is no longer interested, it can then request to unsubscribe itself, whereby it no longer gets updates.

This is implemented within the Pelican as an Interface Manager as shown in Fig. 11. As an example, the inter-

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO2BC004

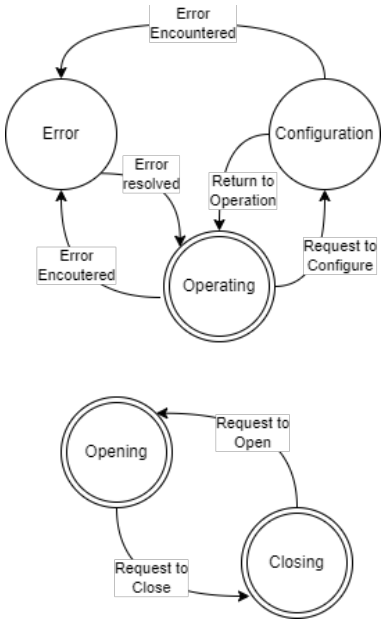

Figure 10: Valve Finite State Machine.

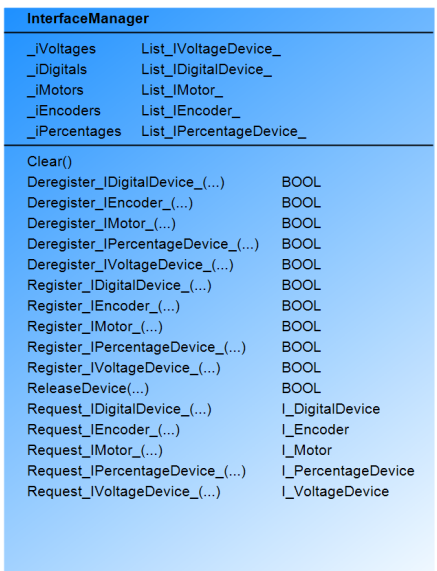

Figure 11: Interface manager.

face manager provides a way for devices on the TcDAL, to register themselves with an object from TcHAL. This way, a device such as a valve, can get the control and feedback signals from the I/O to indicate if an opened or closed limit switch has been activated. Essentially, an object within the TcZookeeper which has implemented the IObservable interface, is able to utilize the Interface Manager in this way.

Legacy Code Integration

To integrate the existing legacy code into the new TcZookeeper suite, it was necessary to develop an adapter to connect the two together. The adapter design was derived from the adapter pattern [9, 11] and in this case, the adapter essentially re-routes the I/O of the hardware from the legacy code, and obtains the value directly from the interfaces within the Pelican. The adapter here performs the below functions:

- Map an existing signal into the appropriate Pelican Interface
- Register each of the new signals with the Interface Manager
- Connect in the hardware signal to the Interface within the adapter code
- Implement methods as part of the adapter to pass the values from the TcHAL into the appropriate interface

As such, the behaviour and functionality of the pre-existing device remains as it is, however with the additional feature set which is provided intrinsically from both the Pelican and TcHAL. Overtime, it would be possible to slowly migrate all of the integrated hardware devices across into the new TcZookeeper. Combined with user training, the two PLC frameworks can co-exist until the legacy code becomes deprecated.

SUMMARY

Software principals have always existed within the software engineering domain to aid clarity, guidance and what is commonly referred to as 'best practice'. As PLCs shift closer to the realm of software, it becomes a logical step to incorporate these principals into the PLC, especially with the increase in memory and the availability of powerful programmatic features which are commonly offered within programming languages. It is noted that while the PLC itself does not implement a true programming language, some adaptions had to be made in order to implement many of the design patterns mentioned.

Aligning the PLC code closer to common software principles results in clean and well structured code. It also ensures that the PLC code is robust, testable and also in a state which can easily be amended as the requirements change over time.

ACKNOWLEDGEMENTS

The PLC team and authors of this paper worked closely with other EuXFEL scientific support groups and acknowledge their continuous efforts, input and cooperation. We thank the rest of the Electronic and Electrical Engineering (EEE) group, the Information Technology and Data Management (ITDM) group and the Controls Software and Data Analysis groups.

REFERENCES

[1] Programmable controllers - Part 3: Programming languages, International Electrotechnical Commission, IEC 61131-3:2013; https://webstore.iec.ch/publication/4552

[2] PLC Programming Then & Now: The History of PLCs, https://www.c3controls.com/white-paper/history-of-programmable-logic-controllers/

[3] Architectural Design Records, https://adr.github.io/

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO2BC004

[4] Mark Richards, "Chapter 1: Layered Architecture", in *Software Architecture Patterns*, O'Reilley Media, Inc. https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html

[5] *NASA Systems Engineering Handbook*, NASA/SP-2016-6105 Rev2, pp. 135-138; https://www.nasa.gov/reference/systems-engineering-handbook/

[6] Test Driven Development, https://www.ibm.com/garage/method/practices/code/practice_test_driven_development/

[7] TcUnit - A TwinCAT unit testing framework, https://tcunit.org/

[8] T. Freyermuth, B. Baranasic, M. Bueno, N. Coppola, L. Feltrin Zanellatto, P. Gessler, *et al.*, "Progression Towards Adaptability in the PLC Library at the EuXFEL", in *Proc. 13th Int. Workshop Emerging Technol. Sci. Facil. Controls (PCaPAC'22)*, Dolní Brežany, Czech Republic, Oct. 2022, pp. 102–106. doi:10.18429/JACoW-PCaPAC2022-FRO13

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns*, Addison-Wesley, 1994.

[10] What is the observer pattern, https://www.ionos.com/digitalguide/websites/web-development/what-is-the-observer-pattern/

[11] Patterns in der Softwareentwicklung: Das Adapter-Muster, https://www.heise.de/blog/Patterns-in-der-Softwareentwicklung-Das-Adapter-Muster-7285630.html

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI