Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

DIPLOMARBEIT

Data Acquisition Software for a Silicon Strip Detector Readout System

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Diplom-Ingenieurs

unter der Leitung von
Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Karl Riedling
Institut für Sensor- und Aktuatorsysteme

eingereicht an der Technischen Universität Wien
Fakultät für Elektrotechnik und Informationstechnik

von
Thomas-Benjamin Obermayer, BSc
Gusenleithnergasse 5/1/11
1140 Wien

Wien, im November 2012 ______________________

In Erinnerung an meinen Grossvater,
der mir beibrachte, wie man den
Commodore 128 programmiert.

In memory of my grandfather:
For teaching me how to program
the Commodore 128.

Kurzfassung

In der Hochenergiephysik werden zur Reproduktion von Teilchen-Trajektorien Siliziumstreifendetektoren eingesetzt. Zur Erfassung entsprechender Detektordaten werden speziell dafür vorgesehene Readout-Chips und Auslesesysteme verwendet.

Das Institut für Hochenergiephysik der Österreichischen Akademie der Wissenschaften (HEPHY) hat einen Prototyp für das Auslesesystem des Belle II Silicon Vertex Detectors vorgestellt, welches Readout-Chips des Typs APV25 initialisiert, steuert und ausliest. Dieses Auslesesystem wird im Rahmen des Belle II Experiments im japanischen Forschungszentrum für Hochenergiephysik (KEK) zum Einsatz kommen. In Ergänzung zum genannten Prototyp wurde eine Windows-Applikation zur Datenerfassung mittels Personal Computer entwickelt. Mängel im Software-Design und in der Dokumentation verursachten Probleme bei der Wartung und Weiterentwicklung dieses Programms.

Diese Arbeit beschreibt den Entwicklungsprozess, das daraus resultierende Design und die Implementierung einer neuen, auf Linux basierenden Applikation zur Erfassung von Daten mit dem genannten Auslesesystem. Eine Anforderungsspezifikation, welche vor der Softwareentwicklung erstellt wurde, wird präsentiert. Ferner werden eine reproduzierbare Vorgehensweise für die objektorientierte Analyse (OOA) und die Einhaltung allgemein anerkannter Grundsätze im objektorientierten Design (OOD) beschrieben. Es wird gezeigt, dass etablierte Entwurfsmuster und Softwareentwicklungstechniken mit dem Ziel verfolgt wurden, ein wartbares C++ Programm zu erstellen, welches einfach zu lesen und testen ist und entsprechend neuer Anforderungen angepasst werden kann. Außerdem beschreibt die Arbeit, wie ein wiederverwendbares Framework aus Softwarekomponenten für das Belle II Experiment erstellt wurde. Die Struktur und das Verhalten der Software werden mit einer geeigneten Teilmenge der Unified Modeling Language (UML) beschrieben.

In ihrer Gesamtheit stellt diese Arbeit einen Ansatz vor, wie etablierte Softwareentwicklungstechniken eingesetzt werden können, um einen vernünftigen Kompromiss zwischen zeitlichem und technischem Aufwand auf der einen Seite und Softwarequalität auf der anderen Seite zu erreichen.

Abstract

It the area of high energy physics, Silicon Strip Detectors (SSDs) are used to reproduce the trajectories of particles traversing a particle detector. The readout of SSDs is accomplished by dedicated readout chips and readout systems intended for Data Acquisition (DAQ).

The Institute of High Energy Physics of the Austrian Academy of Sciences (HEPHY) developed a prototype of the Belle II Silicon Vertex Detector readout system, designed for initialising, controlling and reading out front-end chips of the APV25 type. It will be used for the Belle II Experiment conducted at the KEK research laboratory in Japan. An appropriate Windows-based DAQ software application supporting this prototype was also developed at HEPHY.

However, shortcomings in the design and documentation of this DAQ application caused issues in maintenance and further development.

This thesis describes the development process, the resulting design and the implementation of a new, Linux-based, DAQ application supporting the prototype readout system mentioned above. The thesis also presents a requirements specification, which was finalised ahead of further software development. Furthermore, a reproducible strategy for Object-Oriented Analysis (OOA) and the compliance with generally accepted principles of Object-Oriented Design (OOD) are described. It is shown how established design patterns and software engineering techniques were followed with the aim to create a maintainable C++ program that is easy to read, test and debug, and can be adapted to changing requirements. In addition to this, the extraction of a stable, state-of-the-art framework of components reusable for the Belle II Experiment is illustrated. An appropriate subset of the Unified Modeling Language (UML) is used to describe software structure and behaviour in brief.

As a whole, this thesis presents an approach as to how established software engineering techniques can be utilised to reach a reasonable tradeoff between temporal and technical development effort on the one hand and software quality on the other hand.

Danksagungen

Meine Danksagungen gelten zunächst Herrn Prof. Dr. Karl Riedling, unter dessen engagierter wissenschaftlicher Betreuung diese Arbeit geschrieben wurde. Für ein lehrreiches Jahr danke ich Herrn Dr. Markus Friedl und Herrn Dipl.-Ing. Christian Irmler vom Institut für Hochenergiephysik der Österreichischen Akademie der Wissenschaften. Meinen KollegInnen Annekathrin, Maria, Helmut, Immanuel und Paul danke ich für die gute gemeinsame Zeit während meiner Tätigkeit an diesem Institut. Verpflichtet bin ich Frau MMag. Veronica Peintinger, die in einer für sie bewegten Zeit den Text in mühevoller Arbeit gelesen, korrigiert und damit maßgeblich aufgewertet hat.

Meinen aufrichtigen Dank spreche ich meinen Eltern aus, die mir einen Zugang zu Wissenschaft und Technik ermöglicht und mich während meiner gesamten Studienzeit vertrauensvoll unterstützt haben. Meiner Freundin Marion möchte ich für die mir gegenüber aufgebrachte Geduld danken, aber auch für die liebevolle Unterstützung während der Erstellung dieser Arbeit. Andreas, Gabriela und Mario Marschall bin ich für ihre freundschaftliche Hilfestellung verbunden. Schließlich möchte ich Herrn Mag. Günter Wildmann für zahlreiche Anregungen und Diskussionen danken, welche meine Vorgehensweise nachhaltig bereichert haben.

Contents

Kurzfassung iv

Abstract v

Danksagungen vi

1 Introduction 1

1.1 Issues Addressed in This Thesis 2
1.2 Overview 2
1.3 Typographic Conventions 2

2 The Belle II SVD Readout System 3

2.1 The Hardware Setup 4
2.2 Data Acquisition Software 13

3 Software Requirements 19

3.1 Purpose 20
3.2 Scope 20
3.3 Requirements Format and Numbering 21
3.4 User Requirements Definition 21
3.5 System Requirements Specification 30

4 Software Development Methodology 31
4.1 Object-Oriented Analysis . . . 33
4.2 Object-Oriented Design . . . 35
4.3 Object-Oriented Programming . . . 41

5 Software Design and Implementation 45
5.1 Documentation . . . 46
5.2 Reuse-Oriented Architecture . . . 46
5.3 Third Party Standard Components . . . 46
5.4 VME Interface Abstraction . . . 49
5.5 Readout System Abstraction . . . 53
5.6 Model-View-Controller . . . 56
5.7 Interaction between TuxDAQ and TuxOA . . . 64
5.8 Logging . . . 65

6 Application Overview for Users 69
6.1 Starting TuxDAQ . . . 69
6.2 Different Graphical User Interface (GUI) Tabs . . . 70
6.3 Performing a DAQ Run . . . 71
6.4 VME Bridge Initialisation . . . 73
6.5 FADC+PROC Test . . . 73
6.6 Error Treatment . . . 74
6.7 The Configuration File . . . 74
6.8 Data Output Files . . . 74

7 Software Environment 75
7.1 Operating Systems and VME Libraries . . . 75
7.2 Development Tools . . . 76
7.3 Prerequisites for TuxDAQ Operation . . . 76

8 Conclusion 77

A Traceability 79
A.1 Source Traceability . . . 79

A.2 Requirements Traceability 81
A.3 Design Traceability 83

B The TuxDAQ Repository 87

Acronyms 89

Bibliography 91

1 Introduction

In recent years, a lot of research in the field of high energy physics has focused on readout systems for Silicon Strip Detectors (SSDs) and associated Data Acquisition (DAQ) software. SSDs are used to reproduce the trajectories of particles traversing a particle detector. In the related research area, DAQ systems are products and processes used to collect data produced by a sizeable number of front-end chips. These are attached to the SSDs. Data acquired in this way is stored and provides a basis for further analysis of physical phenomena.

In regards to these systems, the Institute of High Energy Physics of the Austrian Academy of Sciences (HEPHY) has made noteworthy contributions to the Belle Experiment[1] [2] as well as to the proposed follow-up experiment Belle II [3] at the KEK Research Laboratory[2] in Japan.

In 2009, Friedl et al. presented an early prototype readout system for the future Belle II Silicon Vertex Detector (SVD) [4]. This system provides appropriate hardware components for initialising, controlling and reading out front-end chips of type APV25 [5].

Supporting this prototype, a DAQ software application was developed. It was designed to set up all components involved on basis of a configuration file to control the readout system properly, to acquire respective data, to run a limited online analysis[3] on this data and to store it for further offline analysis.

Based on National Instruments (NI) LabWindows/CVI, the program is a proprietary solution. It is tied to Microsoft Windows operating systems. Moreover, its ‘monolithic’ software architecture, which has gradually grown, and the implementation in ANSI C[4] show suboptimal levels of maintainability and portability[5].

Hardware and software have proven successful during experimental research at HEPHY, CERN and at other research facilities [3]. Nevertheless, the aforementioned facts and shortcomings in documentation make the present DAQ program an impractical option for both long-term usage in a developing experimental environment and for reuse in the Belle II Experiment.

[1] The Belle Experiment was explicitly honoured in the statement of the Nobel Prize in Physics 2008 for the experimental verification of the CP violation theory [1].

[2] High Energy Accelerator Research Organisation, Tsukuba

[3] The term ‘online’ is to be understood as ‘during operation’ in this context.

[4] Defined by ANSI X3.159-1989, also referred to as C89

[5] Quality measures as defined in ISO/IEC 9126-1

1.1 Issues Addressed in This Thesis

This thesis describes the development process, the resulting design and the implementation of a new DAQ application for the prototype Belle II SVD readout system.

The new program will be based on the existing solution as far as function is concerned. Nevertheless, an elaborated object-oriented architecture as well as the usage of field-proven principles and patterns will improve the quality measures mentioned above. With C++[6], a widely spread state-of-the-art programming language will be used for implementation. The application will be based on Linux - currently the favoured operating system in the scientific field at hand. Furthermore, the new system will support VME interface adapters by different manufacturers. The online analysis of acquired data will be extracted to a separate program. This will improve runtime flexibility and DAQ performance. Enhanced software modularity and a comprehensive documentation will allow for the reuse of components for the Belle II Experiment and for other projects in the area of high energy physics research. Porting the program to different operating systems in future will be easier, and design changes in the hardware will cause fewer software modifications than before.

In this thesis I am presenting an approach to how established software engineering techniques can be utilised to an extent that builds a reasonable tradeoff between development effort (temporal and technical) and software quality. An appropriate subset of the Unified Modeling Language (UML)[7] is used to describe the software structure and behaviour.

1.2 Overview

In Chapter 2, the proposed Belle II readout system is described from a software developer's perspective. The requirements for the new DAQ application can be found in Chapter 3. A description of the software development methodology in Chapter 4 is followed by a detailed overview of the software design and implementation in Chapter 5. An application overview for users is presented in Chapter 6. Chapter 7 describes the software environment of the new DAQ program. Finally, Chapter 8 provides a comprehensive summary and describes what future work should include.

1.3 Typographic Conventions

Text in *italic face* represents a term that is defined within this thesis. Subsequent instances of such a term are not in italic face. **Bold face** letters emphasise respective content and serve to improve readability.

Binary numbers are tagged with a subscript 'b', e.g. 11110101_b. In contrast, hexadecimal numbers are tagged with a subscript 'h', e.g. $F5_h$. Numbers without any subscript are decimals, e.g. 245.

All-numeric calendar dates have the format of *dd/mm/yyyy*.

[6] Defined by ISO/IEC 14882:2003
[7] Defined by the UML Specification 2.0, 2006 (Object Management Group)

2

The Belle II SVD Readout System

This chapter describes an early prototype of the Belle II SVD readout system, as available at HEPHY. The system was originally designed for an intermediate upgrade of the Belle SVD2, which was never implemented [3]. This prototype, however, contains all types of components of the planned new productive system albeit on a smaller scale. In the Belle II Experiment, the implementation of more SSDs will require a higher effort of readout hardware than provided by the present system. Thus, some design modifications will be necessary for the final design [6]. These modifications will potentially require adaptations of the software involved.

In the course of prototype development, an appropriate DAQ software application has evolved. Although this program has represented an elaborated and useful tool for several beam tests[8] and lab tests, it will only be suitable for future requirements to a limited extent (see Chapter 1).

Nevertheless, from a functional point of view, the existing program provides a state-of-the-art basis for further software development. Hence, a brief introduction to this software application will follow a description of the hardware setup.

[8]These tests (typically conducted at CERN) are intended for the examination of SSD prototypes with a high-energy particle beam and all the equipment involved (e.g. readout electronics, cooling, software).

2.1 The Hardware Setup

As shown in Fig. 2.1, the hardware setup at HEPHY consists of the following components[9]:

- A radioactive source (default: ^{90}Sr) emitting beta-particles (electrons) which are to be investigated. In contrast to a high energy beam, this source is available and suitable for lab tests at HEPHY.
- A *scintillator* with an attached *photomultiplier and trigger unit*, which - in combination - deliver a trigger pulse (referred to as *hardware trigger*) when a traversing particle is detected. Installed behind one or more detector modules, these devices indicate particles that have crossed all the sensors. Such an incident is also referred to as an *event*.
- Up to eight so-called *hybrids*, each of them attached to an SSD (single-sided or double-sided) for position measurements of traversing particles. Depending on the types of sensor and hybrid board, these *front-end electronics* usually contain four to ten *readout chips* of type APV25.
- One *Dock box* containing a motherboard (*Mambo*), which hosts two *Repeater Boards (REBOs)*
- A VMEbus[10] system (*VME crate*) containing *back-end electronics*. These contain one *VME controller*, one readout master controller (*NECO*[11]) along with a control distribution unit (*SVD3_Buffer*[12]) and two *Flash Analogue-to-Digital Converter and Data Processing (FADC+PROC) boards*.
- A *Personal Computer (PC)* connected to the VME controller via an integrated *adapter card*. This PC runs an appropriate software application for the acquisition of data, referred to as *data acquisition software*.

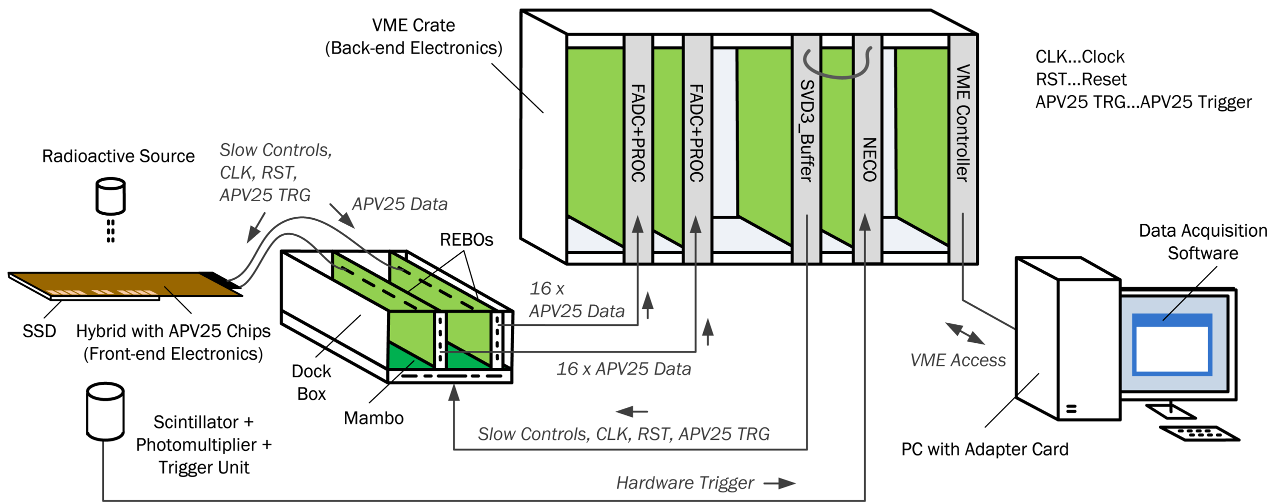


Figure 2.1: *Prototype of the Belle II SVD Readout System at HEPHY*

[9] Power supplies and respective cabling are not considered.
[10] Defined by ANSI/IEEE 1014-1987
[11] NECO is an acronym for New Controller.
[12] SVD3 is an acronym for Silicon Vertex Detector 3, which was originally designed as an upgrade of the Belle SVD2.

On the following pages, the main electronic components of the readout chain will be explained from a software developer's perspective. Referring to Fig. 2.1, the readout chain will be described from left to right, starting with the SSDs.

2.1.1 Silicon Strip Detectors

A Silicon Strip Detector (SSD) is a semiconductor device intended for the detection of traversing charged particles and photons, respectively. It is also referred to as *sensor* and comprises a number of doped narrow strip implants, which act as charge collecting electrodes. Placed on one side of a low-doped silicon bulk, these strips build a one-dimensional array of diodes. By connecting each strip to a charge-sensitive amplifier (readout chip), one-dimensional position measurements of traversing particles can be achieved. Respective detectors are referred to as Single-Sided Silicon Strip Detectors (SSSDs).

Two-dimensional position measurements can be achieved with so-called Double-Sided Silicon Strip Detectors (DSSDs). These detectors provide another set of doped narrow strips on the backside of the bulk.

Fig. 2.2 describes the function principle of an AC-coupled n-type bulk DSSD[13]. The strips, together with the bulk, provide a set of diodes, which are reverse-biased. That is, n^+-strips are connected to a voltage more positive than the voltage of the p^+-strips. The bias voltage causes an electric field $\vec{E}$ within the semiconductor.

If a charged particle traverses the device, it produces electron-hole pairs (due to energy loss). Under the influence of the electric field, the carriers move towards respective electrodes (strips). The drifting carriers induce currents $i(t)$ in the electrodes, which can be acquired by AC-coupled charge-sensitive amplifiers.

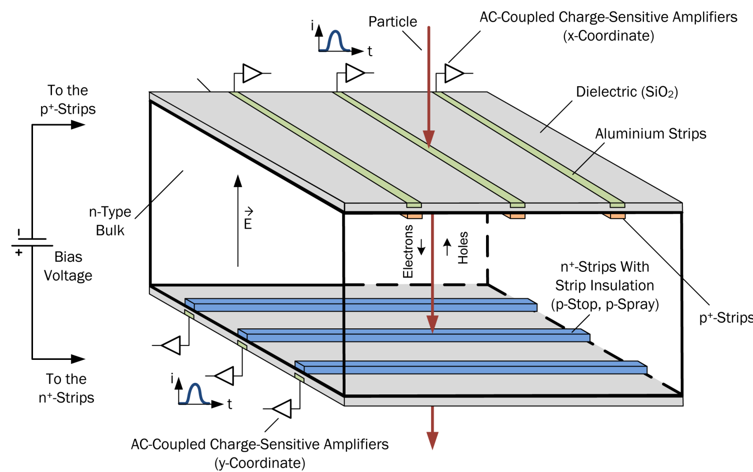


Figure 2.2: *Function Principle of an n-Type Bulk Double-Sided Silicon Strip Detector (DSSD)*

The original Belle II SVD design contains 187 n-type bulk DSSDs, each providing either 512 or 768 n^+-strips and 768 p^+-strips [3]. These design parameters may change up until the completion of the final design. At HEPHY, both SSSDs and DSSDs with various strip configurations are used for experimental research.

[13] This type of DSSD is used in the Belle II Experiment.

2.1.2 APV25 Readout Chips

The APV25 readout chip is an analogue pipeline Application-Specific Integrated Circuit (ASIC) designed for the readout of 128 strips of an SSD. Fig. 2.3 presents a block diagram of the chip's internal structure.

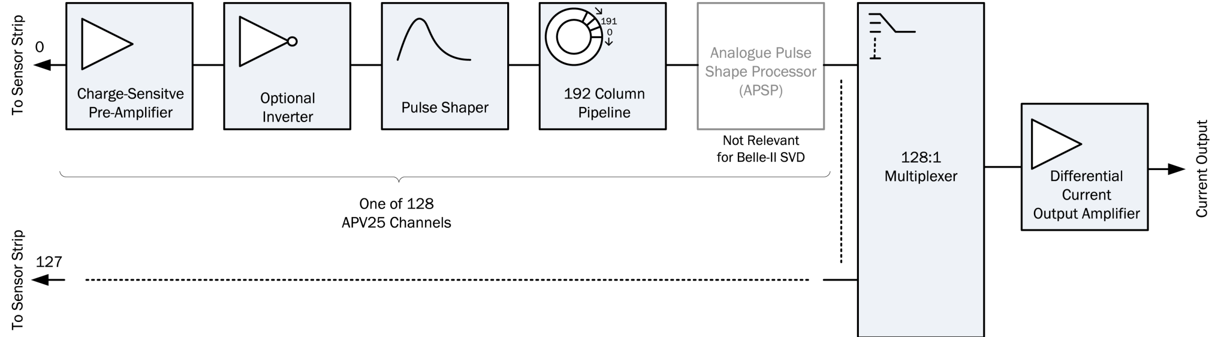


Figure 2.3: *APV25 Block Diagram, Based on Fig. 1 of [5]*

The ASIC provides 128 channels, each dedicated to the readout of a specific sensor strip. Within a channel, the charge values obtained from the corresponding strip are amplified, optionally inverted, shaped and periodically written to a 192 column analogue ring buffer memory. This memory is referred to as *pipeline*, and it allows practically dead-time free measurements [4]. Depending on the operation mode, either one (*peak mode*) or more (*multi-peak mode*) consecutive samples of one pipeline column can be read out[14].

These samples are obtained from the pipeline and output by a multiplexer stage once an appropriate trigger signal is given. From now on, this trigger signal will be referred to as *APV25 trigger* (in contrast to the hardware trigger signal introduced in Section 2.1). Two consecutive APV25 triggers must be separated by at least two clock cycles [7]. This constraint defines the *maximum rate of APV25 triggers*.

The APV25 has a differential analogue current output where the 128 multiplexed samples of analogue strip data are represented by current values[15]. Due to the internal multiplexer structure, the order of channels read out is non-consecutive [7]. In the output signal of one respective sample, the sequence of 128 analogue values is preceded by a digital header, a digital 8-bit address and an error bit. The address refers to the pipeline column containing the analogue values of when the chip was triggered. A synchronisation pulse (*tick mark*) is output periodically, regardless of whether an APV25 trigger has been given or not[16]. Fig. 2.4 shows the analogue output format of the APV25.

A slow control interface conforming to the I^{2}C standard[17] allows the control of certain circuit parameters and other general settings (e.g. the operation mode). The chip also provides an internal calibration feature, which is outlined in [8].

As per original plan, the implementation of 1902 APV25 chips into the future Belle II SVD is proposed [4]. However, a concrete number of instances will be defined when the detector design is frozen. The prototype at HEPHY was designed for 384 chips. Because of limitations in installed readout hardware, it can be used for the readout of 32 chips.

[14]For *hit-time reconstruction* (implemented on FADC+PROC boards), typically six consecutive samples are taken [4].

[15]$g_{AS} = 1\,\mathrm{mA}/MIP$, g_{AS}...APV25 Analogue Signal Gain, MIP...Minimum Ionisable Particle, see Fig. 5.4 of [8]

[16]Subsequent to the sequence of 128 analogue values of the last sample output before the chip changes to idle mode. In idle mode, a tick mark is issued every 35 clock cycles.

[17]Defined by the I^{2}C-Bus Specification, Version 2.1, January 2000 (NXP Semiconductors)

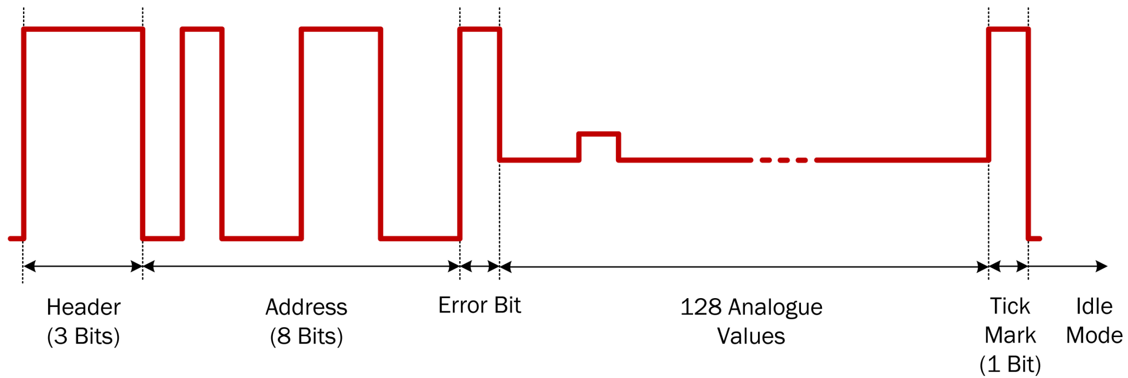


Figure 2.4: *APV25 Analogue Output Format (Abscissa: Time; Ordinate: Reconstructed Amplitude of Differential Output Current), see [7]*

2.1.3 Hybrids

The APV25 requires only a limited amount of external wiring. Among electrical measures, the chip's I^2C address is programmed by tying dedicated pins to ground or leaving them floating. Furthermore, the arrangement of multiple readout chips on one composite module means that the readout of entire sensors can be accomplished (typically containing 512 strips or more per side).

For that purpose, so-called hybrid boards have been developed. The future Belle II SVD design proposes two different types. In addition to *conventional hybrids* [8], *Origami chip-on-sensor modules* [9] will be implemented.

Fig.2.5 (left) shows a conventional hybrid board attached to a SSSD. It contains three of four possible readout chips serving 384 sensor strips. The readout of a Double-Sided Silicon Strip Detector (DSSD) would require two conventional hybrid boards. The photo on the right shows the Origami chip-on-sensor concept, where ten readout chips are mounted on top of a DSSD as part of a flexible electronic circuit . Electrically, four readout chips belong to one circuit serving the sensor's n-side (512 strips), whereas the other six chips belong to a separate circuit serving the p-side (768 strips).

For electrical reasons, one readout circuit can only serve one respective sensor side (either n-side or p-side). Therefore, a DSSD requires at least two separate readout circuits. For the sake of simplicity, I will refer to an entire hardware readout module as *hybrid* (irrespective of its type). By contrast, I will refer to the electrical circuit serving one sensor side as *detector module*. In the case of Origami modules (serving DSSDs), one hybrid maps to two detector modules (one containing four readout chips and the other one containing six readout chips). However, a conventional hybrid maps to one detector module (containing four readout chips). Each detector module comprises a separate I^2C control bus system[18].

A design upgrade allowing for up to six readout chips per conventional hybrid board has been proposed. Considering this upgrade, the Belle II SVD design would require the implementation of 89 Origami modules and of 169 conventional hybrids [3]. This would map to 1902 APV25 chips spread over 347 detector modules. These numbers have to be considered as tentative up to the completion of the design process. The prototype at HEPHY can read out eight detector modules of four readout chips each.

Hybrids containing less than the maximum number of readout chips are also possible.

[18]By default, I^2C addresses 34, 36, 38, 40 (42, 44) are dedicated to the APV25S-1 chips of one respective detector module.

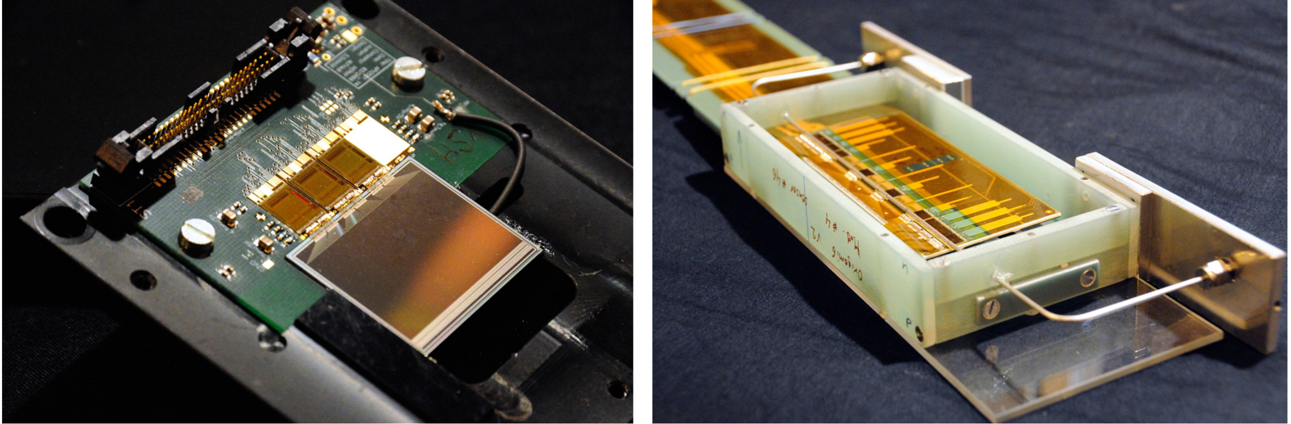


Figure 2.5: *A Conventional Hybrid Board (left) in Contrast to a Prototype of the Origami Concept (right)*

2.1.4 Dock Box

The front-end detector modules have to be provided with clock, reset, APV25 trigger and I^{2}C control signals (*slow controls*), all of which are generated and delivered by the NECO master controller / SVD3_Buffer in the back-end (see Section 2.1.5). Moreover, analogue data obtained from detector modules is to be transferred to dedicated FADC+PROC boards (also located in the back-end, see Section 2.1.6).

Not only do all of these signals have to be buffered, but in addition to that, floating low voltage signals of the front-end have to be translated to ground-bound signals of the back-end and vice versa [3]. These tasks are accomplished by repeater boxes (Dock boxes) located a few meters away from the front-end electronics. Electronically, each Dock box contains a motherboard (Mambo) hosting up to six Repeater Boards (REBOs).

Fig. 2.6 gives a schematic overview of both components: The motherboard is composed of two electrical subunits, Mambo #0 which hosts up to three REBOs dedicated to n-side detector modules and Mambo #1 hosting up to three REBOs dedicated to p-side detector modules[19]. Each REBO slot has a unique address within one Mambo subunit.

Actual signal buffering and level translation are implemented in the REBO units. In these units, clock, reset, APV25 trigger and slow controls obtained from the back-end are translated and subsequently delivered to respective detector modules. Furthermore, data read out from the front-end is translated and transported to the back-end.

The prototype readout chain at HEPHY was designed for the operation of up to four Dock boxes, altogether containing eight Mambo subunits and 24 REBOs. Every REBO can serve up to four detector modules, each of which contains a maximum of four readout chips.

Dock boxes will be replaced by *junction boxes* in the proposed final design. These junction boxes will only contain connectors, voltage regulators and passive components. Together with the FADC+PROC functionality, signal level translation will be implemented into a new composite back-end board, which will serve up to 24 readout-chips[20]. About 80 of these boards (spread over several VME crates) will be deployed [3].

[19]These assignments are specific to the HEPHY lab; in any other setting Mambo #0 could also be dedicated to p-side modules and Mambo #1 to n-side modules.

[20]Either six detector modules of four chips each or four detector modules of six chips each

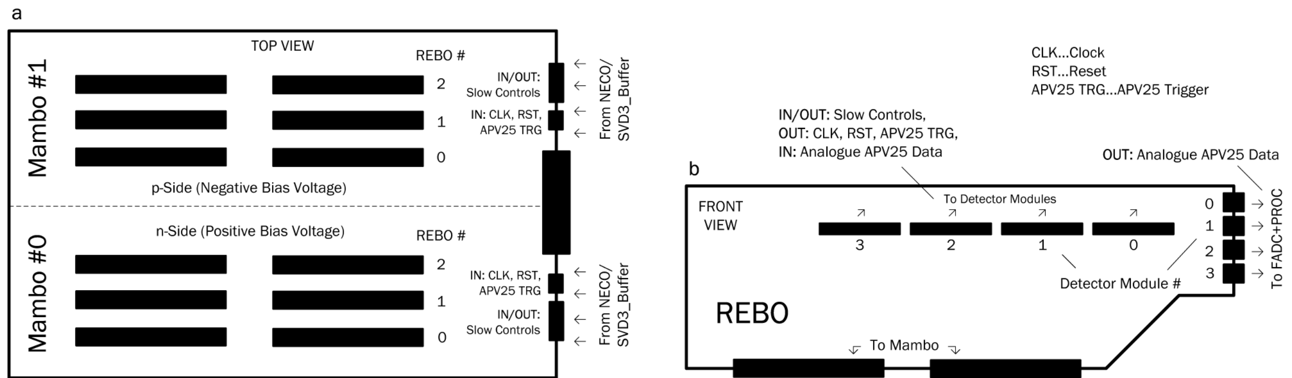


Figure 2.6: *Schematic Overview and Addressing Schemes of (a) Mambo and (b) REBO*

2.1.5 NECO Master Controller and SVD3_Buffer

The New Controller (NECO) is a 9U VME board[21] located in the back-end. It can be controlled by software via the VMEbus interface. To achieve front-end configuration and flow-control, the NECO unit allows to address detector modules following the addressing schemes of Mambo and REBO (see Fig. 2.6). It delivers slow controls from the back-end to Dock boxes, where they are forwarded to connected detector modules. Within a detector module, readout-chips are addressed via a dedicated I^{2}C bus (see Section 2.1.3). In addition to slow controls, the board generates and delivers clock, reset and APV25 trigger signals.

Fig. 2.7a gives an elementary overview of the NECO master controller. It contains two Field Programmable Gate Arrays (FPGAs). FPGA<DB>comprises several logical components, among them:

- Several registers for enabling, resetting and configuring involved equipment
- A so-called *VETO logic*, which limits the rate of allowed hardware triggers to the maximum rate of APV25 triggers (see Section 2.1.2). In addition to that, it ensures that data processing and DAQ can be accomplished in real-time.
- An *APV25 trigger and reset logic* for the generation of appropriate APV25 trigger and reset signals

FPGA<VME> is intended for VMEbus communication. In addition, this FPGA comprises an *I^{2}C master unit* for the utilisation of I^{2}C busses on NECO, on REBOs and in the detector modules.

The VETO logic blocks all hardware triggers exceeding a configurable number of *allowed triggers*[22]. A *VETO flag* is set when the number of allowed triggered has been reached. This flag can be read out and cleared by software.

Allowed triggers are passed to the APV25 trigger and reset logic. This unit generates APV25 triggers and APV25 reset signals conforming to the chip's specifications. In contrast to hardware triggers, that are allowed to pass through by the VETO logic, the generation of APV25 triggers[23] can be initiated by software. These *software triggers* are, however, asynchronous to actual events at the sensor. In Fig. 2.8, an overview is given. Different trigger types are marked blue, and software initiated actions are represented in green colour.

[21] Default VME address of NECO prototype at HEPHY: $1A000000_h$
[22] Typically one (configurable by configuration file)
[23] And APV25 reset signals, respectively

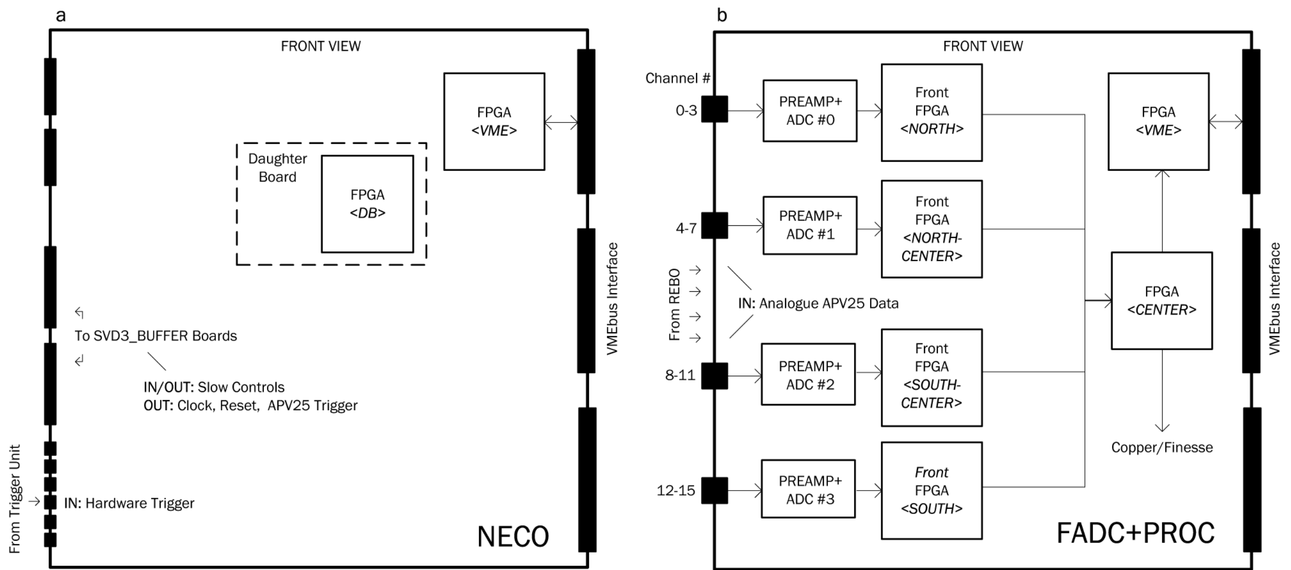


Figure 2.7: *Schematic Overview and Addressing Schemes of (a) NECO and (b) FADC+PROC*

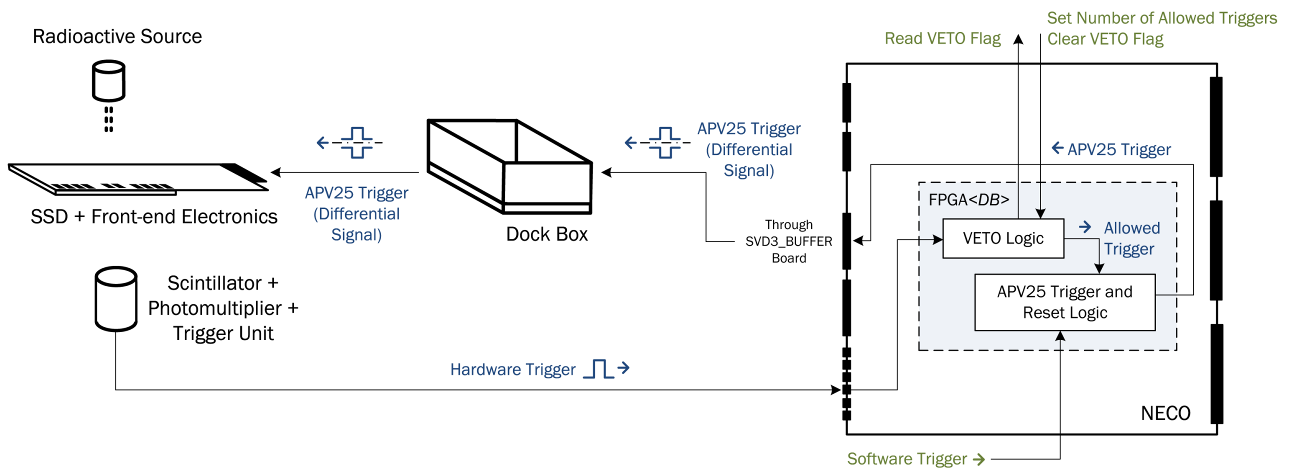


Figure 2.8: *VETO Logic, APV25 Trigger and Reset Logic of NECO*

Propagation delays of involved signals (mainly caused by cable lengths) are non-negligible. To achieve correct Analogue-To-Digital Converter (ADC) timing and synchronisation of clock and trigger signals, delay elements[24] are implemented on the NECO and FADC+PROC boards as well as in the REBO units. Providing an I^2C interface, the delay elements of NECO and REBO can be configured by the NECO I^2C master unit.

The design of the prototype readout chain would allow to spread back-end electronics over two VME crates. In this scenario, the NECO master controller, installed in one crate, would be supported by two SVD3_Buffer distribution boards (each buffer board located in a separate VME crate). The task of these distribution boards is to buffer signals and to forward them to Dock boxes accordingly.

In the present prototype system the NECO master controller could be connected to two separate SVD3_Buffer boards, each allowing the connection of two Dock boxes. For the final design, an upgrade, which conforms with the modified structure mentioned in Section 2.1.4, will be necessary.

2.1.6 FADC+PROC Data Processing Boards

Like NECO, the Flash Analogue-to-Digital Converter and Data Processing (FADC+PROC) unit is a 9U VME board[25] in the back-end. A conceptual overview is given in Fig. 2.7b. The board provides 16 inputs for analogue APV25 data bundled to groups of four readout chips. These bundles correspond to specific detector modules and consequently to their dedicated REBO outputs. Data of all channels is processed and subsequently held available for data acquisition.

Each input is connected to an adjustable equaliser, which compensates for involved frequency-dependent cable loss. This unit is followed by a preamplifier and a 10-bit Flash Analogue-to-Digital Converter (FADC). In subsequent FPGAs (referred to as *front FPGAs*), groups of four channels have a dedicated pipelined processing unit which can perform channel reordering (to restore the physical strip order, see Section 2.1.2), pedestal subtraction, a two-pass common mode correction and zero suppression (sparsification). Following this, the data from all channels is collected, formatted and buffered in FPGA<CENTER>(referred to as *central FPGA*) [4]. The board can be operated in two different modes:

1. *Transparent mode* All raw APV25 data can be acquired as it has been obtained from the front-end (in non-consecutive strip-order). This data is held available in First In, First Out (FIFO) memories[26], each of which is assigned to an input channel of the board and implemented in the Front FPGAs.

2. *Hit mode* Different from transparent mode, a filtered subset solely containing significant data of particle hits can be acquired. This filtering is done by the hardware in real-time. The resulting hit data from all channels is collected, re-ordered and stored in a FIFO memory[27] of the central FPGA.

Data can be acquired via the VMEbus interface (implemented in FPGA<VME>). This option is the simplest, and therefore used at the HEPHY lab. Alternatively, a COPPER/FINESSE interface could be used. This is a “general purpose pipelined readout platform developed at KEK” [3]. It would allow data readout at a higher transfer rate but would come with additional hardware requirements.

[24] Of type Delay25 [10]

[25] Default VME addresses of prototype FADC+PROC boards at HEPHY: $1B000000_h, 2B000000_h$

[26] Namely *FIFO1*

[27] Namely *FIFO3*

The prototype readout system was designed for the use of 32 FADC+PROC data processing boards. For the final design of the Belle II SVD readout system, the deliberations in Section 2.1.4 have to be taken into consideration.

2.1.7 VME Controller and VME/PC Interface

The NECO master controller and the FADC+PROC data processing units in the back-end can be accessed via their VMEbus interface. For this purpose, a PC containing an adapter card is connected to a VME controller, which is installed in the VME crate. A PC software program can use this combination (referred to as *VME bridge*) to set up and control a DAQ process.

At HEPHY lab, an NI MXI-card for the conventional PCI[28] bus in combination with an appropriate NI VME-MXI-2 VME crate controller module has been used for several years. This solution has become outdated for the following reasons:

- In modern PC equipment, the conventional PCI bus interface has become deprecated. Sticking to a PCI-based adapter card would make the selection of new computer equipment difficult.
- Experience has shown that the stability of software drivers and according software programs is suboptimal.

Therefore, two alternative VME bridges were acquired:

1. CAEN A2818 PCI adapter [11] and A3818 PCIe[29] adapter [12] / V2718 VME controller [13]
2. Struck Innovative Systeme (SiS)1100e PCIe adapter / 3104 VME controller [14]

Both VME bridges are characterised by fibre-optic links and the availability of Linux-based software drivers. They promise improvements on the outdated NI solution and are the first choice for all future activities.

Fig. 2.9 shows the situation at HEPHY lab. In total, three different VME controllers with appropriate adapter cards are available.

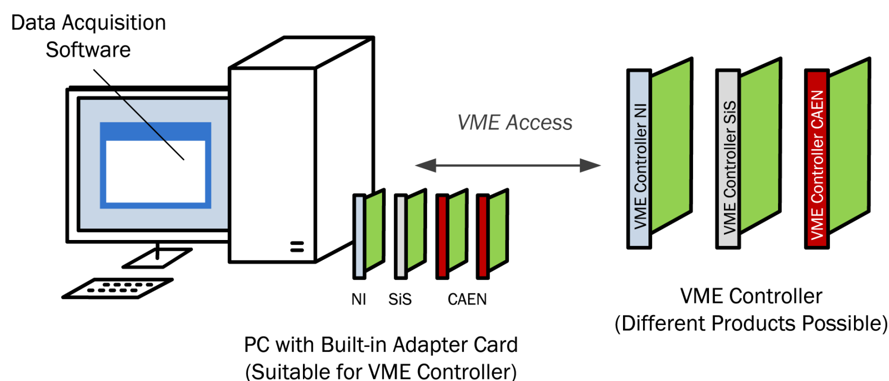


Figure 2.9: *VME Bridges at HEPHY Lab*

[28]Defined by the PCI Local Bus Specification, 2002 (PCI-SIG)
[29]Defined by the PCI Express 1.0a Specification, 2003 (PCI-SIG)

2.2 Data Acquisition Software

The readout system described in Section 2.1 is an enhancement of an APV25-based predecessor system named *APVDAQ* [15]. An appropriate DAQ software application [8] had been deployed with APVDAQ before both hardware and software were extended gradually in the course of time.

Therefore, the DAQ software program for the present prototype readout system provides many functions and algorithms of the original APVDAQ system. These are supplemented by extensions addressing new hardware (NECO, FADC+PROC, Mambo, REBO) and respective Graphical User Interface (GUI) elements. Taking this background into account, the evolved program is still referred to as *APVDAQ readout application*.

It was developed with NI LabWindows/CVI, an event-driven ANSI C Integrated Development Environment (IDE) that provides engineering-specific functionality for instrument control, DAQ, analysis and User Interface (UI) development [16]. As mentioned in Chapter 1, the application is Windows-based.

2.2.1 Functional Overview

The APVDAQ readout application is used to manage DAQ runs by means of the prototype Belle II SVD readout system. In this context, a *DAQ run* can be understood as a process designed to obtain particle detector data from a set of front-end readout chips. Acquired data can be stored for further analysis by an offline analysis tool, which has also been developed at HEPHY (see Section 7.4 of [8]). The APVDAQ readout application fulfils five major tasks:

1. Definition of the hardware structure and setup of configuration parameters based on a *configuration file* (see Section 2.2.5)
2. Appropriate initialisation of all components involved
3. Flow control management by proper delivery of VME messages to NECO and FADC+PROC boards and delivery of slow controls to the front-end (via NECO)
4. Data acquisition from FADC+PROC boards and storage of acquired data
5. Online processing and analysis of acquired data

2.2.2 Run Types

The APVDAQ readout application provides several types of DAQ runs. Irrespective of the type, a given number of events[30] are always captured at the beginning of a run. For these events (referred to as *initial events* or *pedestal events*), software triggers are initiated periodically by the program in order to determine pedestals and signal noise of all sensor strips[31] observed. Resulting values are used for appropriate signal correction by means of hardware (FADC+PROC) and software.

[30] Typically 600 (parameter in the configuration file)

[31] Under the assumption that the random noise process at the sensor is time-invariant within the software trigger interval, periodical (deterministic) sampling and stochastic sampling provide same results. Therefore, the acquisition of pedestal events based on periodical trigger delivery can be considered as a random process.

Among others, the APVDAQ readout application provides the following DAQ run types:

- *Hardware run* This is used for source measurements in the laboratory as well as for beam tests. At the beginning, all hardware components are initialised based on a given configuration file. In the next step, the acquisition of pedestal events is accomplished. Subsequently, the status of the VETO flag (see Section 2.1.5) is checked periodically at the pace of the DAQ timer. If the VETO flag is set, new event data is pending. Consequently, data acquisition, processing and storage (if desired) are accomplished before the program clears the VETO flag again. This procedure is repeated until the desired number of events have been captured. Fig. 2.10 visualises the hardware run in the form an UML activity diagram.

- *Software run* In contrast, this process is entirely based on software triggers. The VETO logic is disabled, and consequently any hardware triggers are ignored. In a software run, software triggers are initiated cyclically at the pace of the DAQ timer. As software triggers are asynchronous to particle hits, this run type is not suitable for the acquisition of relevant particle data. However, the advantage of a software run is that no radioactive source or beam is needed. Therefore, it can be used for development purposes and to test the proper functioning of the whole readout chain.

- *Run for internal calibration* This utilises the built-in calibration feature of the APV25 chip. It is used to check the peaking time and to obtain calibration constants for each strip channel. Further offline analysis tools require this data for converting ADC values into electron-based signals [8].

- *ADC delay scan* This allows the user to determine the best setting of the ADC delay, which is associated with each readout channel of the FADC+PROC board. The ADC delay value has an essential impact on the timing of respective FADC chips.

For a detailed description of both the calibration run and the ADC delay scan, see Section 6.6.1 of [8][32].

2.2.3 Performing a DAQ Run

Fig. 2.11 shows the GUI of the APVDAQ readout application. Below, the typical scenario for performing a DAQ run[33] is described. The numbers and letters between parentheses refer to the respective labels in the figure.

After starting up the application, the GUI is loaded. Now the user can select a configuration file (1), which defines the structure of the readout hardware (front-end and back-end) as well as certain configuration parameters. Having selected an appropriate file, the arrangement of front-end electronics defined within it may be verified with a geometry display. This tool can be opened with (2). If the given configuration settings need to be varied, the configuration file can be edited during runtime with an external text editor. Then, the modified file can be reloaded (3) without a restart of the application.

In the next step, the user can select the type of DAQ run that is to be performed (4). Depending on whether or not acquired data should be written to a file, an output path, a run name and respective comments can be entered (5).

[32] Explanations in [8] about a common clock delay for all readout channels do not apply. FADC+PROC boards allow to define this value for each particular channel.

[33] Hardware run or software run

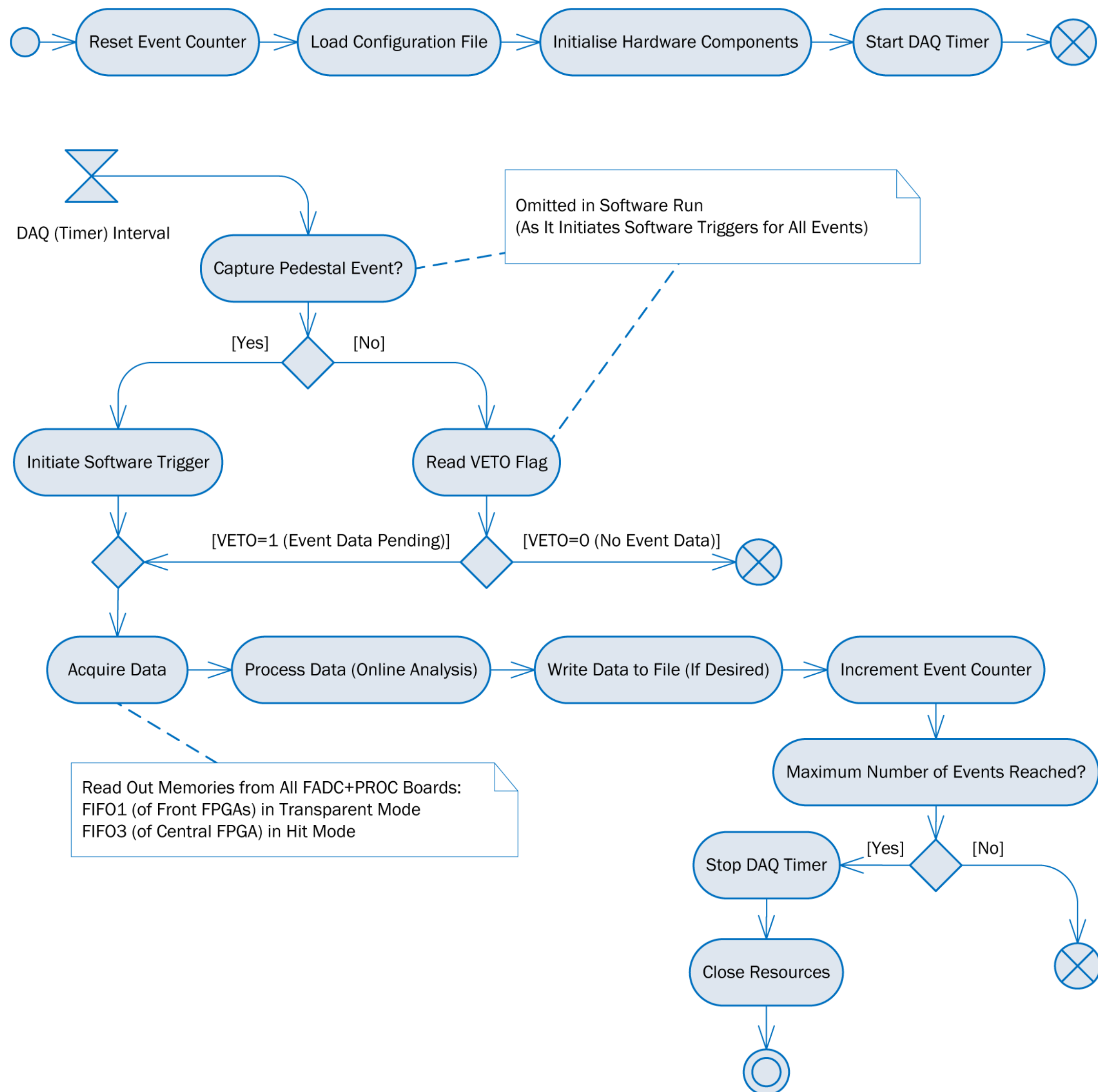


Figure 2.10: *Hardware Run of the APVDAQ Readout Application (UML Activity Diagram, Reverse-Engineered)*

Once an appropriate configuration file has been loaded, a run type selected and data output set up properly, a DAQ run can be initiated with the 'Start Run' button (6). Following this, the application performs an initialisation of all involved hardware components based on configuration file settings. For this purpose, it addresses NECO and FADC+PROC boards via the VME bridge. Slow controls, intended for the front-end, are handed over to the NECO controller, which distributes them to Dock boxes accordingly. They are forwarded to respective detector modules. This procedure can be traced in a console window (7), which is also used for error notifications and other status messages.

Once the setup of hardware components has finished, the acquisition of data from the FADC+PROC boards starts. In addition to the console window, several GUI widgets (8) monitor the ongoing DAQ run. Further to the actual APV25 trigger rate, the number of the currently captured event, the number of occurred APV errors and the Estimated Time of Arrival (ETA) (estimated completion time) are presented to the user.

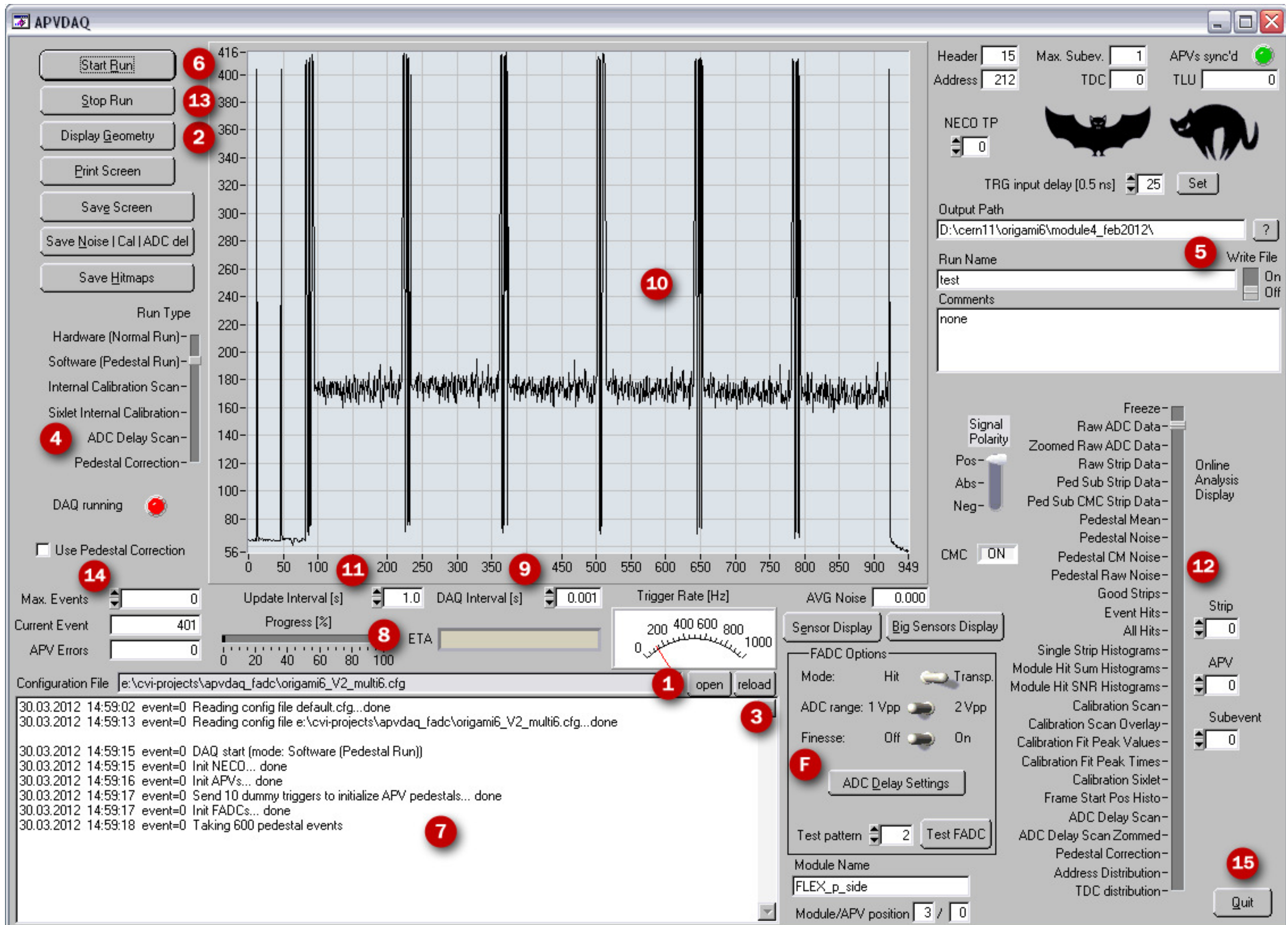


Figure 2.11: *GUI of the APVDAQ Readout Application*

A timer-based architecture allows the user to interact with the GUI during a DAQ run. Therefore, a *DAQ interval*[34] can be defined (9), which corresponds to a period of time during which data is acquired cyclically. A plot widget (10) provides graphical information on the recently captured and processed data[35]. The plot is updated periodically at an interval[36] that can be adjusted separately (11). Here, a value of zero would lead to plot updates at the pace of data acquisition.

The user can choose among various plot types by means of a dedicated slider widget (12), and analysis results are presented in the plot widget [37]. The algorithms provided allow noise calculation, pedestal subtraction, common mode correction and hit finding.

Generally, a DAQ run concludes after the 'Stop Run' button (13) has been pressed. In addition to that, the user may define a maximum number of events to be captured (14). In this case, the DAQ run terminates

[34]Typically (and at minimum) 1 ms

[35]Extraction of raw strip data from data frames and reordering of sensor strips (see Section 2.1.2)

[36]Typically 1 s

[37]In Fig. 2.11, the option 'Raw ADC Data' is selected (12): Raw FADC data currently read out from one specific APV25 front-end chip is visualised by the plot widget (abscissa: time in clock periods; ordinate: amplitude in ADC counts). The plot shows the data regarding one respective event, where six consecutive samples have been captured in consequence of one APV25 trigger signal (chip operated in multi-peak mode). As mentioned in Section 2.1.2, for each of the six samples a digital header is followed by 128 values of analogue strip data. In the example at hand, analogue values hint at channel-to-channel pedestal variations rather than a significant particle hit on the sensor. In contrast, Fig. 5.28 of [3] shows a comparable plot containing data of a captured particle hit.

automatically as soon as the desired set of events has been recorded. This setting can also be changed during an ongoing DAQ process. If the value is zero, the DAQ run does not conclude automatically. Instead, the user has to terminate the process manually.

All acquired data is written to disk, if desired (see Section 2.2.6). The APVDAQ readout application allows the execution of several successive DAQ runs without the need for a program restart. The user can quit the application by pressing the 'Quit' button (15).

2.2.4 FADC+PROC Test

In addition to several run types, the APVDAQ readout application provides a test function for FADC+PROC boards (see Fig. 2.11, (F)).

Different *test patterns* (predefined test data) can be loaded into a dedicated test Random Access Memory (RAM) of an FADC+PROC unit. Then, the test function acquires this data from the board via VME bridge and plots both test data as well as acquired data on the screen. If both plots match, it can be assumed that the tested unit works properly.

By these means, FADC+PROC hardware and firmware, VME connectivity and applied settings can be tested. Note that for such a test the checked FADC+PROC board has to be configured for autonomous clock generation [17].

2.2.5 The Configuration File

Ahead of a run, an appropriate configuration file has to be loaded. This configuration file contains:

- *Definition and configuration of front-end electronics* This implies different APV25 parameters, the definition of composite detector modules and their addressing within the given hardware structure.
- *Definition and configuration of back-end electronics* In addition to the addressing of all back-end boards, specific parameters for NECO and FADC+PROC boards can be defined.
- *DAQ parameters* These are hardware and software parameters, which are relevant for a data acquisition process.
- *Sensors and zones configuration* For online analysis purposes, sensors can be defined. In addition to the definition of their physical parameters, sensors are associated with readout chips. Furthermore, they can be partitioned into several zones[38].
- *Other parameters* These imply parameters for calibration and hit recognition.

2.2.6 Data Output Files

Depending on the type of a run, the APVDAQ readout application produces several output files. During a DAQ run, the following files are produced:

- *A copy of the original configuration file (.cfg)* As the used configuration file contains essential information on a specific run, a copy is created. This copy is used by offline analysis tools.

[38] A *sensor zone* can be considered as a unique set of strips of a specific sensor.

- *A log file (.txt)* This is a plain text file, which contains the run type, the run name, respective comments, the number of events, a note on the original configuration file and logging output that has been produced during a run. The log file is also used by offline analysis tools.
- *Binary data files (.dat)* Data acquired from FADC+PROC during a DAQ run is stored in binary format. If a large amount of data is acquired, a new data file is spawned after every 2 GB. The generated files serve as a data basis for offline analysis.

The run for internal calibration produces a *calibration file (.cal)*, which is described in Section 6.6.1 of [8]. The ADC delay scan does not produce any data output files [8].

3

Software Requirements

Chapter 1 outlined the role of the APVDAQ readout application at HEPHY and the demand for a new concept to meet future requirements. In Chapter 2, the hardware environment and the APVDAQ readout application were described briefly.

This chapter presents the software requirements for a new DAQ application. The provided requirements were identified in a dedicated *requirements engineering process*, which included the following activities:

- *Requirements elicitation* Professional staff was accompanied and interviewed during everyday work with the APVDAQ readout application at HEPHY lab and during a beam test at CERN in October 2011. In so doing, typical *scenarios* were identified.
- *Requirements classification, organisation, prioritisation and negotiation* These activities were mainly performed in the context of *weekly meetings* with the staff of the HEPHY Department of Electronics.
- *Iterative requirements definition and specification* This was accomplished with the help of a *Wiki* system[39] [18] for documentation and communication.

The organisation of this chapter follows the recommendations of IEEE 830-1998 [19] and [20]. Sections 3.1 to 3.3 serve as a brief introduction. Section 3.4 provides a User Requirements Definition (URD) containing (*functional* and *non-functional*) *user requirements*, which describe the services provided by the new application from a user's point of view. As a comprehensive System Requirements Specification (SRS) would be out of scope of this thesis, an exemplary overview in Section 3.5 shows how formal *system requirements* can be defined. These build a solid basis for the design, implementation and test of the required application.

[39] A web-based content management system which is characteristically easy to handle

All presented methods of requirements engineering have been chosen with the claim for a balance between effort and benefit. They are suitable for engineers of all disciplines, who develop software in an environment of non-computer scientists. Therefore, formal methods that can be learned easily were chosen over more difficult ones whenever their use was considered reasonable or inevitable.

3.1 Purpose

This chapter describes the user requirements for a new DAQ program that is suitable for the prototype of the Belle II SVD readout system (see Chapter 2).

The intended audience is the staff of the institute's Department of Electronics and successional students, who will use and advance the system, the thesis advisor Dr. Karl Riedling and the developer.

3.2 Scope

Section 2.2.1 described the main tasks of the APVDAQ readout application. The software product to be developed will be functionally based on this program, but it will assign the stated tasks to two separate Linux-based programs:

- Tasks 1 to 4 will be covered by a program named *TuxDAQ*[40]. The present thesis deals with this program and gives a comprehensive overview on the requirements, design and implementation of TuxDAQ.
- Task 5 will be covered by another program, the *TuxOA*[41]. A detailed description of that program is beyond the scope of this thesis. However, many of the introduced components can be used for both TuxDAQ and TuxOA. An outlook on the interaction between both programs is presented in Section 5.7.

Both programs will exchange acquired data via a dedicated interface. The separation of tasks promises an increase of runtime flexibility and performance scalability. Fig. 3.1 shows the new configuration in a UML component diagram.

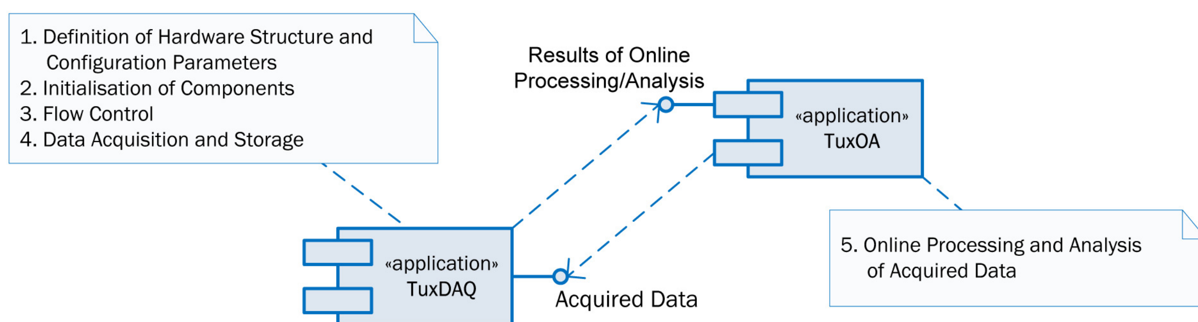


Figure 3.1: *Separation of Tasks (UML Component Diagram)*

[40]The name TuxDAQ is a composition of *Tux* and *Data Acquisition (DAQ)*. Tux is the name of the offical Linux mascot [21].
[41]The name TuxOA is a composition of *Tux* and *Online Analysis (OA)*.

3.3 Requirements Format and Numbering

The verb *shall* is used whenever the described requirement is mandatory, *should* is used for desirable requirements. Sentences without any of these verbs are explanations. Text belonging to formal user requirements is in **bold** face to improve readability.

User requirements are identified by the section they belong to and also by an item number. This item number is unique within a particular section. For references to a specific requirement, a *user requirement identifier* is used, following the syntax rule below:

Requirement <section>/<number>, where

<section>	is the number of the section to which the requirement belongs to
<number>	is the item number used with each instance of <section>.

If a requirement is referenced within a particular section, the section number and the separating slash are omitted.

3.4 User Requirements Definition

This section characterises the typical TuxDAQ user and describes the services which TuxDAQ offers for them. In addition to user requirements (sentences containing 'shall' or 'should'), explanations are provided.

As a framework of maintainable components for the Belle II Experiment will be extracted, special attention has been given to several kinds of *traceability* [20]:

- *Source traceability* User requirements are linked to the people who proposed them (see Appendix A.1) and supported by their rationales. In so doing, system developers can understand requirements better, and qualified people can be consulted in case of a requirements change.

- *Requirements traceability* By documenting dependencies between requirements, it is possible to evaluate the number of affected requirements when one particular requirement changes. See Appendix A.2 for a detailed requirements traceability overview.

- *Design traceability* Links between requirements and design modules allow system developers to identify all design modules impacted by a proposed requirements change. For a design traceability table, see Appendix A.3.

It should be mentioned that the user requirements presented here were primarily formulated with a focus on the user perspective and readability. Although they have the characteristics of good requirements[42], user requirements are neither specific enough to serve as a straight basis for programmers nor are they suitable for the identification of software test cases. For these scenarios, specific system requirements are needed (see Section 3.5).

[42]Requirements should be correct, unambiguous, complete, consistent, ranked according to importance, verifiable, modifiable and traceable [19].

3.4.1 User Characteristics

Fig. 3.2 gives an overview on the TuxDAQ user hierarchy.

Typically, a *TuxDAQ user* is a physicist or a an electronic engineer who is educated in the field of SSDs. They are interested in acquiring data and its according interpretation. The typical TuxDAQ user understands where to connect hybrid prototypes and how to operate the readout system and software application. Relevant configuration files are provided to the typical TuxDAQ user. In case of any error, the typical TuxDAQ user consults a specialist for troubleshooting.

A *specialised TuxDAQ user* is a developer, who has a sophisticated understanding of the readout system. In case of problems (in hardware or software), they are responsible for troubleshooting. This user type knows the implemented software procedures at least on a system level, and they are able to create configuration files for different hardware setups.

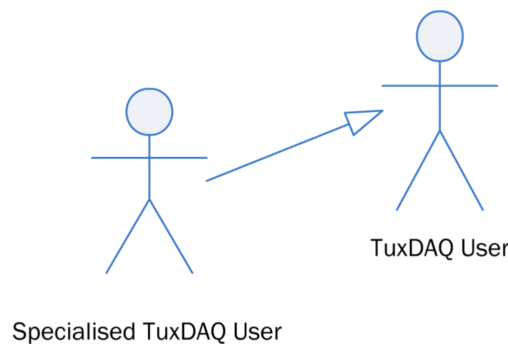


Figure 3.2: *TuxDAQ User Hierarchy (UML Actors and UML Generalisation Relationship)*

3.4.2 TuxDAQ Environment

The (hardware and software) environment of TuxDAQ can be described as follows:

1. Compatibility with the Belle II SVD Prototype Readout System:

   TuxDAQ shall be a DAQ application that is suitable for the prototype of the Belle II SVD readout system, which is described in Chapter 2.

2. Multiple VME Bridges:

   Like the APVDAQ readout application, TuxDAQ will run on a PC that comprises an adapter card connected to a matching VME controller (installed in a VME crate). This so-called VME bridge represents a *low-level communication interface* between the back-end and the DAQ system. In contrast to the APVDAQ readout application, **TuxDAQ shall operate with different kinds of VME bridges (see Section 2.1.7) to communicate with the readout system.** Fig. 2.9 illustrates the existing VME environment at HEPHY.

3. Interoperability with HEPHY Offline Analysis Tool:

   As a successor of the APVDAQ readout application, TuxDAQ is embedded in an environment of various established tools and processes. In particular, compatibility of TuxDAQ with the existing offline analysis

tool at HEPHY is important. To achieve this, **TuxDAQ shall produce the same set of data output files as the APVDAQ readout application (see Section 2.2.6)**. This refers to file types, names, sizes and formats.

4. Linux Compliance:

   Due to the prevalence of Linux in the related scientific field, **TuxDAQ shall run on Linux-based operating systems**.

3.4.3 Field of Application

1. Main Purpose of TuxDAQ:

   TuxDAQ, together with the readout system, shall acquire data from SSD prototypes at the HEPHY laboratory in Vienna.

2. Usage at External Research Facilities:

   In addition to lab tests, TuxDAQ shall serve as DAQ application for beam tests at different external research facilities. Typically, HEPHY staff organise periodical beam tests of new prototype equipment at CERN and KEK. For these beam tests, the readout system is installed at the external sites.

3. Reusability for the Belle II Experiment:

   a) **The architecture and implementation of TuxDAQ shall allow software developers to extract a framework of maintainable components for the Belle II Experiment.**

   b) **These components shall cover tasks 1, 2 and 3 of Section 3.2.** For task 4 (acquisition of data), a different dedicated DAQ system (including appropriate DAQ software) will be developed [3].

3.4.4 Configuration

1. Configurability by Means of a Configuration File:

   a) **TuxDAQ shall allow for the definition of the hardware structure of the readout system by means of a configuration file that can be loaded by the user.** This does not include any settings specific to VME bridges.

   b) **TuxDAQ shall allow for the definition of parameters for the configuration of components by means of the same configuration file.** This does not include any settings specific to VME bridges.

   c) **Referring to this configuration file, TuxDAQ should keep the legacy configuration file format of the APVDAQ readout application (see Section 2.2.5).** Consequently, extra effort and expense for the re-creation of existing configuration files and staff training can be avoided.

2. No Sensors and Zones Necessary:

   TuxDAQ shall not mandatorily require any configuration parameters for sensors and zones (as provided by legacy configuration files). This is because no online processing or analysis of the acquired data will be performed by TuxDAQ (see Requirement 3.4.5/2).

3. Selection of VME Bridge:

   TuxDAQ shall allow the user to select a VME bridge for operation prior to program execution.

3.4.5 Main Functions

TuxDAQ shall carry over functionality from the APVDAQ readout application as follows:

1. Exploration of Loaded Configuration:

   TuxDAQ shall allow exploration of a loaded configuration. This could be achieved by the visualisation of both the hardware structure and the parameters defined in a configuration file. The exploration of a configuration by a skilled person prior to a run can prevent system malfunction.

2. No Online Data Processing and Analysis:

   a) **Irrespective of the run type, TuxDAQ shall not perform any processing or analysis of the acquired data during operation.**

   b) **If data storage is desired, TuxDAQ shall store these data in its original format as acquired from the FADC+PROC boards.**

   This is because all online processing functionality will be extracted to TuxOA (see Section 3.2).

3. Hardware Run:

   a) **TuxDAQ shall allow for a hardware run as described in Section 2.2.2.**

   b) **From a functional point of view, this run type shall be identical to the hardware run of the APVDAQ readout application except for the constraint mentioned above (see Requirement 2).**

4. Software Run:

   a) **TuxDAQ shall allow for a software run as described in Section 2.2.2.**

   b) **From a functional point of view, this run type shall be identical to the software run of the APVDAQ readout application except for the constraint mentioned above (see Requirement 2).**

5. Typical Workflow:

   a) **Based on the scenario described in Section 2.2.3, TuxDAQ shall allow the user to perform hardware and software runs according to the typical workflow outlined in Fig. 3.3.**

   b) **TuxDAQ shall allow activities prior to 'Perform Run' not only in the depicted sequence, but also in an arbitrary order (e.g. selecting a run type could be performed before a configuration file is loaded, etc.).**

   c) **TuxDAQ shall allow the user to request an 'infinite run', which captures events until they stop the run (or until the output data carrier is full).**

   d) As an alternative, **TuxDAQ shall allow the user to define a maximum number of events to be captured (in this case, the program shall terminate the run automatically).**

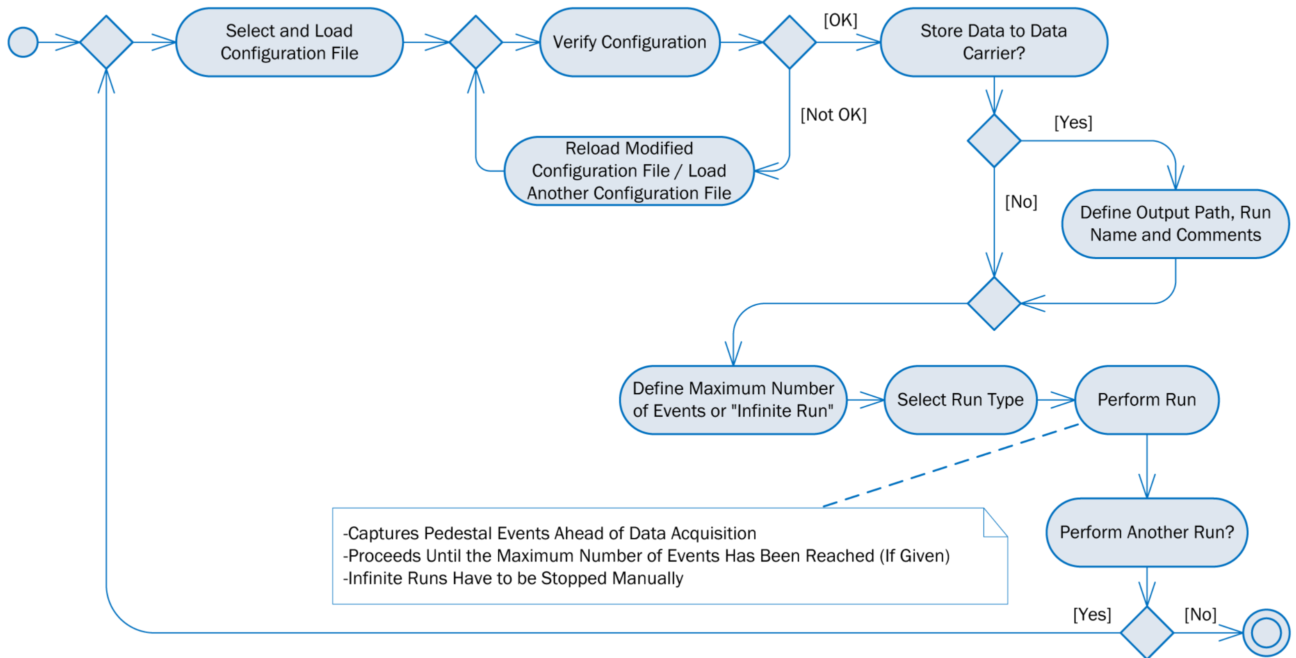


Figure 3.3: *TuxDAQ Requirement: Typical Procedure of Performing a Hardware Run or a Software Run (UML Activity Diagram)*

e) **TuxDAQ shall allow the user to modify both settings, even during a run.**

f) **If during a run a user sets the maximum number of events lower than the current number of events, TuxDAQ shall ignore the (invalid) user input.**

g) **In addition to the activities depicted in Fig. 3.3, TuxDAQ shall allow the user to define a DAQ interval (see Section 2.2.3).**

6. Run for Internal Calibration:

   a) **TuxDAQ shall provide a run for internal calibration, as described in Section 2.2.2.**

   b) The APVDAQ readout application stores data to a .cal file. This data has been processed after its acquisition. In contrast, **TuxDAQ shall store data to a .dat file that has been acquired directly from the FADC+PROC boards.** This is due to the fact that all online processing functionality will be extracted to TuxOA (see Section 3.2).

7. ADC Delay Scan: **TuxDAQ shall provide an ADC delay scan, as described in Section 2.2.2.**

8. FADC+PROC Test:

   a) **TuxDAQ should provide an FADC+PROC test function, which is based on the test function of the APVDAQ readout application (see Section 2.2.4) as far as function is concerned.**

   b) **TuxDAQ should allow the user to test any channel of any configured FADC+PROC board.**

9. Progress Monitoring of Active Run:

   a) **TuxDAQ shall allow the user to follow the progress of an ongoing run, so that it is possible for the user to monitor the proper functioning of the readout system during operation.**

   b) **TuxDAQ shall also provide additional information for specialised TuxDAQ users.** As a result, they can track the proper functioning of particular subprocesses (e.g. the initialisation of specific components).

10. Estimation of Run Conclusion:

    TuxDAQ shall allow the user to estimate when an ongoing run will conclude. This enables staff to direct their attention to other tasks during long-term runs.

11. Offline Run Reproduction:

    TuxDAQ shall allow specialised TuxDAQ users to reproduce a successful run after conclusion and program termination. On one hand, this helps staff to reproduce proper operation after long-term runs. On the other hand, it may facilitate understanding offline analysis results.

3.4.6 Data Exchange with TuxOA

1. Stand-alone Operability:

   If neither online processing nor online analysis of acquired data are necessary for operation, TuxDAQ shall allow the user to operate it as a stand-alone application, without TuxOA running at the same time. This brings the benefits of runtime-flexibility and performance scalability.

2. Automatic TuxOA Invocation:

   a) **If TuxDAQ requires results of online processing, it shall invoke TuxOA automatically.**

   b) **If this is the case, all data shall be exchanged between both applications automatically.**

3. Collaboration with TuxOA:

   a) **TuxDAQ shall allow the user to request the online analysis of acquired data prior to operation.**

   b) **In this scenario, TuxDAQ shall invoke TuxOA automatically and provide TuxOA with the data acquired.** Online processing and analysis of acquired data are then performed automatically by TuxOA.

   c) **TuxDAQ should also allow the user to request the online analysis of acquired data during operation.**

3.4.7 Error Treatment

1. Configuration File Validation:

   a) **After a configuration file has been loaded by the user, TuxDAQ shall validate this file.**

   b) **If the file is invalid, TuxDAQ shall reject the configuration file.**

   c) **In this case, TuxDAQ shall provide an error notification containing information about the fault for the user.**

   This prevents an undefined malfunction of the readout system due to configuration faults.

2. User Input Validation:

   a) **If the information provided by the user prior to a run or FADC+PROC test is insufficient or invalid, TuxDAQ shall not start the run.**
   b) **Instead, TuxDAQ shall provide an error notification for the user.**

3. Low-level Communication Link Error Handling:

   a) **If the low-level communication link between TuxDAQ and the readout system cannot be established or used properly, TuxDAQ shall abort operation.**
   b) **In this case, TuxDAQ shall provide an error notification for the user.**

   These error notifications on communication link issues may simplify troubleshooting.

4. DAQ Error Handling:

   a) **If, during operation, data cannot be acquired from the readout system properly, TuxDAQ shall abort operation.** Due to the fact that the readout system could have run into an undefined error state, aborting operation is a safe coping strategy.
   b) **In this case, TuxDAQ shall provide an error notification for the user.**
   c) **If available, TuxDAQ shall provide information on the fault as well.** This may simplify troubleshooting.

5. Data Storage Error Handling:

   a) **If, during operation, data cannot be written to the data carrier properly, TuxDAQ shall abort operation.**
   b) **In this case, TuxDAQ shall provide an error notification for the user.**
   c) **If available, TuxDAQ shall provide information on the fault as well.**

   See Requirement 4 for a rationale.

6. Recovery from Failures:

   If any of the aforementioned problems (see Requirements 1, 2, 3, 4 and 5) have occurred, TuxDAQ shall recover in such a manner that the user can start over without a restart of the application. This improves usability and prevents an undefined malfunction during operation.

7. Error Notifications:

   All possible error notifications issued by TuxDAQ shall be expressed in such a manner that specialised TuxDAQ users may identify a fault within a maximum duration of $\mathbf{d_{IF,max}} = 30\,\mathrm{min}$. As

TuxDAQ is not a consumer application, this requirement represents a reasonable tradeoff between development effort and user convenience.

8. Stack Trace on Error:

   For all possible error notifications, TuxDAQ shall present an according *stack trace*[43]. This potentially simplifies troubleshooting.

9. Error Information on the Screen:

   TuxDAQ shall present error notifications and the according stack trace on the PC screen during operation. This allows the user to react on problems immediately.

10. Offline Error Tracking:

    TuxDAQ shall archive error notifications and the according stack trace for an offline error analysis. This is particularly useful when an error has occurred during a long-term run.

3.4.8 User Interface

1. Graphical User Interface (GUI):

   TuxDAQ shall provide a Graphical User Interface (GUI). This is for user convenience.

2. GUI Tasks:

   The TuxDAQ GUI (see Requirement 1) shall provide means to fulfil the following Requirements: 3.4.3/1, 3.4.4/1, 3.4.5/(1, 3, 4, 5, 6, 7, 8, 9, 10), 3.4.6/3 and 3.4.7/9.

3. Console Interface:

   a) **As an alternative to a GUI, TuxDAQ should provide a console interface, which allows the user to operate the application on the standard console of the operating system.** A console interface would allow operation without the need for a graphical desktop system. Additionally, it simplifies process automation. Moreover, some users prefer a console interface to a GUI.

   b) **The console interface should fulfil the same requirements as the GUI does.**

   c) **Moreover, Requirement 3.4.7/8 should be considered.**

4. GUI as Default User Interface:

   a) **By default, TuxDAQ shall load the GUI at program start.**

   b) **TuxDAQ shall provide means that allow the user to declare at program start whether they want to use the console interface as an alternative to the GUI.**

   c) **TuxDAQ shall not provide both user interfaces at the same time.**

[43] A report on the sequence of nested active procedures at the point of time when an error occurred

3.4.9 System Properties and Constraints on the System Implementation

1. TuxDAQ as Open Source Project:

   TuxDAQ shall be available as open source software.

2. Usage of Free Software Components:

   Any programming libraries involved in TuxDAQ development shall be free software. That is, they shall permit combination and redistribution with TuxDAQ for scientific purposes at no cost.

3. Object-Oriented Design and Implementation:

   The TuxDAQ design and implementation shall be based on the object-oriented paradigm. This is a state-of-the-art approach and improves maintainability (see Requirement 3.4.3/3).

4. C++ as Programming Language:

   TuxDAQ shall be programmed in C++. This decision is based on the following reasons:

   - C++ allows to follow the object-oriented paradigm.
   - C++ is wide-spread in the related research area.
   - Application Programming Interfaces (APIs) of the relevant VME bridges are available for C/C++.
   - A variety of C++ tools (compiler, libraries, etc.) are available as free, Linux-based software.
   - Legacy code of the APVDAQ readout application is available in C. Using C++ as programming language allows to carry over C code fragments with minimal effort.

5. Compatibility With the CERN ROOT Framework:

   TuxDAQ shall allow to integrate the CERN ROOT framework. This free, C++ based framework provides a variety of functions related to particle physics and mathematics. Consequently, this requirement might become relevant in the future.

6. Minimum Performance:

   Given that it has the same hardware setup and the same configuration parameters, TuxDAQ shall show at least the same DAQ performance as the APVDAQ readout application. In this context, DAQ performance can be measured in terms of the *effective event rate*[44].

7. Maximum File Size:

   a) **Any data output files produced by TuxDAQ shall not exceed a size of** $2\,\mathrm{GB}$**.**
   b) **If more data is to be produced, several files of up to** $2\,\mathrm{GB}$ **shall be created** (see Requirement 3.4.2/3).

8. System Evolution:

   The software developer(s) of TuxDAQ shall take anticipated changes due to the evolution of the readout system into consideration. These changes are described in Sections 2.1.2 to 2.1.7.

[44] Events captured per second

3.4.10 Documentation

1. Application Programming Interface (API) Reference Manual:

   In addition to this thesis, the developer(s) of TuxDAQ shall provide an API reference manual (with respect to Requirement 3.4.3/3), which serves as a detailed documentation of the TuxDAQ program source code. This improves maintainability.

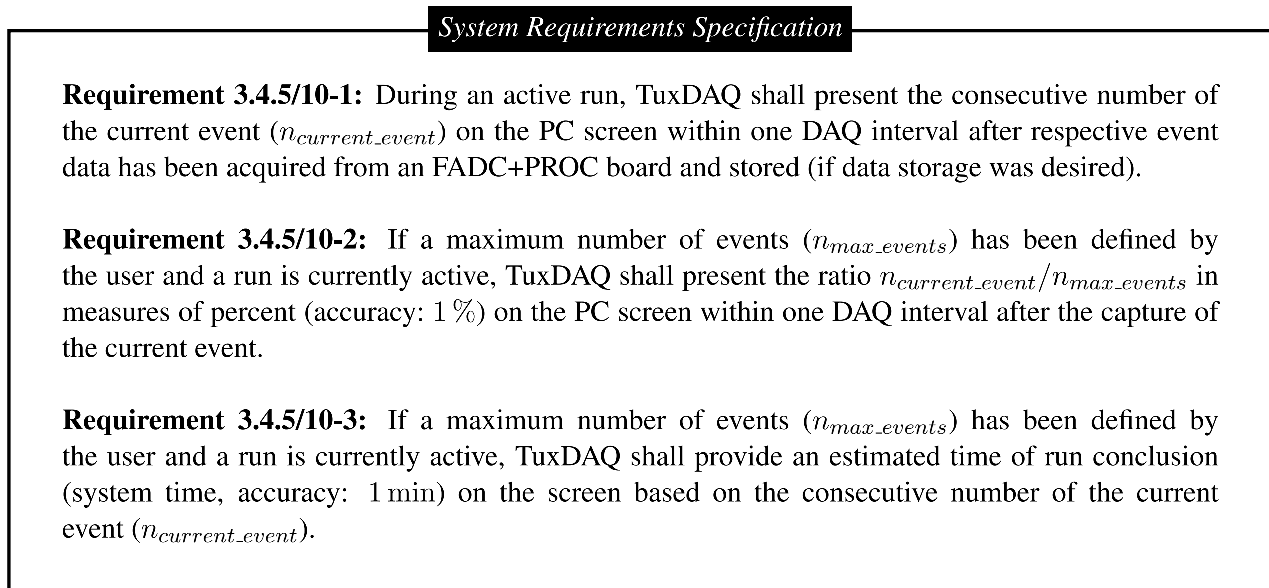
2. Inline Source Code Comments:

   The developer(s) of TuxDAQ shall document complex program artefacts with adequate inline source code comments. Consequently, successive programmers are provided with helpful information for maintenance or further development.

3.5 System Requirements Specification

System requirements build the baseline for system design, implementation and the creation of test cases. Compared to user requirements, they are "more detailed descriptions of the software system's functions, services and operational constraints" and "should define exactly what is to be implemented" [20].

A System Requirements Specification (SRS) would consist of a variety of system requirements. For the sake of compactness, only one example is presented. It demonstrates how a user requirement may expand to several verifiable system requirements. Fig 3.4 shows the system requirements derived from User Requirement 3.4.5/10.



System Requirements Specification

Requirement 3.4.5/10-1: During an active run, TuxDAQ shall present the consecutive number of the current event ($n_{current_event}$) on the PC screen within one DAQ interval after respective event data has been acquired from an FADC+PROC board and stored (if data storage was desired).

Requirement 3.4.5/10-2: If a maximum number of events (n_{max_events}) has been defined by the user and a run is currently active, TuxDAQ shall present the ratio $n_{current_event}/n_{max_events}$ in measures of percent (accuracy: $1\,\%$) on the PC screen within one DAQ interval after the capture of the current event.

Requirement 3.4.5/10-3: If a maximum number of events (n_{max_events}) has been defined by the user and a run is currently active, TuxDAQ shall provide an estimated time of run conclusion (system time, accuracy: $1\,\mathrm{min}$) on the screen based on the consecutive number of the current event ($n_{current_event}$).

Figure 3.4: *System Requirements Derived From User Requirement* 3.4.5/10

Instead of a natural language specification of system requirements, other methods may be used, e.g. Requirements Diagrams and Tables of SysML[45].

[45] Defined by the Systems Modeling Language Specification 1.2, 2010 (Object Management Group)

4

Software Development Methodology

Chapter 3 presented the user requirements for *TuxDAQ*, the new DAQ software application to be developed for the prototype of the Belle II SVD readout system. This chapter provides an overview of the software development process for TuxDAQ.

As already mentioned, TuxDAQ will be based on the APVDAQ readout application (see Section 2.2) as far as function is concerned. As a first step, it was therefore necessary to reach a clear understanding of this application and its role in the whole system as a basis for new software development. This was achieved by two major activities:

- *Interviews with the software developers of the application*[46] These interviews were conducted on a weekly basis to help understand both the prototype readout system and the software application. For documentation and communication purposes, a Wiki system [18] was used.
- *Reverse engineering of the existing program source code* In the course of comprehensive code inspection, diagrams of both the software structure and algorithms were created. Due to shortcomings in documentation, this was a wide-ranging task.

[46] Namely Christian Irmler and Markus Friedl

Due to the requirements for TuxDAQ no architectural concepts from the APVDAQ readout application were carried over. However, algorithms for VMEbus communication, electronics control, DAQ control, etc. and data file properties (types, names, sizes and formats) could be reused.

As the next step, a dedicated software development process[47] was defined, allowing to create a completely new, object-oriented architecture. As shown in Fig. 4.1, TuxDAQ has been developed in an *iterative process*. This is a popular development cycle by which all process steps are performed iteratively so that the software program evolves incrementally. Each process step has a certain outcome, which is refined in every iteration.

The complexity of the Belle II readout system makes it difficult to reach a detailed understanding in a short time. Therefore, an iterative development process, which allows for incremental development, was the best method of choice.

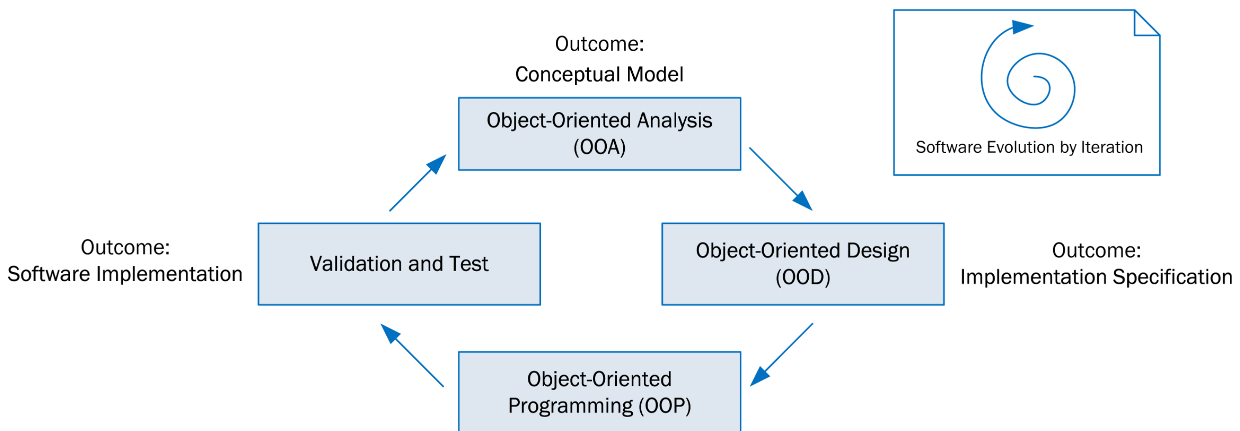


Figure 4.1: *Iterative Process: Software Analysis, Design, Implementation and Test for TuxDAQ*

Within this process, the following activities were performed:

- *Object-Oriented Analysis (OOA)* In this process step, both the prototype readout system at HEPHY and the requirements for TuxDAQ were analysed in regards to the identification of class/object candidates, their attributes and behaviour. As a result, a *conceptual model* was produced, which evolved in every iteration of the development process.

- *Object-Oriented Design (OOD)* The conceptual model was revised to achieve the right granularity, to define appropriate interfaces and inheritance hierarchies and to establish reasonable relationships among the classes. The design's aim was to be specific to the TuxDAQ case of application but general enough to address the goal of extracting a framework reusable for the Belle II Experiment. Each process iteration produced a more elaborated *implementation specification*.

- *Object-Oriented Programming (OOP)* Programming had to include a tradeoff between achieving a high level of software quality (as required for a software framework) and taking into account the learning curve for new developers. These are typically non-computer scientists with profound C++ skills.

- *Validation and Test* Classes were separated by the risk a software error would cause. According to the risk identified, the testing strategy was adapted for several groups of classes. In fact, unit testing was

[47] This process is based on the ideas of [22] and was adapted to the circumstances at HEPHY lab.

only done for critical framework classes. Non-critical classes were investigated by methods of system testing. A modest testing strategy was considered reasonable. The outcome of this process step was the *implementation* of TuxDAQ, which grew incrementally with every iteration.

In the following sections, the methodologies of the respective process phases are described in more detail.

4.1 Object-Oriented Analysis

Due to the complexity of the Belle II SVD readout system, the identification of appropriate class/object candidates for a software model seemed a challenging task. In the early project phase, lack of understanding of the readout hardware made a software-focused analysis of the system even more difficult. To cope with this challenge, an appropriate analysis method, based on the concept of Class Responsibility Collaboration (CRC) cards [23], was developed. This is an established method, and perfectly suitable for non-computer scientists concerned with object-oriented software development.

As shown in Fig. 4.2, a template for forms was created to provide the basis for a detailed description of all components of the readout system. Forms using this template do not only facilitate a better understanding of the underlying platform, but also contain useful information for the extraction of class/object candidates and for documentation purposes. The template provides the following fields:

- *Component* This field is reserved for the name of the described component. That name already indicates a potential *candidate for a class* with the same or a slightly modified name. For example, for the APV25-S1 readout chip, a class candidate named APV25 could be identified.
- *Latest Version/Identifier* Here, the latest version or another identifier of the component is stated. This is useful for the assessment of system evolution issues and for documentation management.
- *Bibliography* This is intended for references to literature describing the component. By gathering this information, it is possible to obtain a good overview of the state of the art. Additionally, having all references at hand saves time in the development process.
- *Number of Pieces in Prototype System* Here, the number of instances of the component in the prototype readout system is stated. For example, there are two FADC+PROC boards available (see Section 2.1.6).
- *Number of Pieces in Belle II SVD* This field contains the number of pieces of the component proposed for the Belle II SVD readout system. Together with the information on available instances in the prototype system, this helps to identify the occurring relationships and their multiplicities. For example, the prototype at HEPHY can read out up to eight detector modules *containing* four APV25 readout chips each, whereas in the final design 347 detector modules, which contain either four or six chips (see Section 2.1.3), are proposed. Based on this information, the following conclusion can be reached: there will potentially be eight to 347 instances of a class candidate *DetectorModule*, each of which is aggregated/-composed (based on the word ‘containing’) of four to six *APV25* instances.
- *Description* Here, the component is described in brief and natural language. Its *responsibilities* are identified, and *collaborations* with other components are discovered. This is not only helpful to gain

Component

Latest Version/Identifier

Bibliography

Number of Pieces in Prototype System | Number of Pieces in Belle II SVD

Description

Relevant Interfaces, Connectors, Inputs, Outputs

Timing Issues

Parameters Configured by APVDAQ Readout Application
Configuration Formats/Protocols

Data Acquired by APVDAQ Readout Application
Acquisition Formats/Protocols

Error States

Potential Future Modifications

Miscellaneous / Questions

Figure 4.2: *Form Template for a Systematic Object-Oriented Analysis of the Belle II SVD Readout System*

an understanding of the system but also allows a lingustic analysis of this information. Consequently, class candidates can be associated with *nouns*, relationships and operations with *verbs* and attributes with *adjectives*. There have been a lot of publications on similar techniques, e.g. [24], and these methodologies have proven popular and successful [25] in practical experience.

- *Relevant Interfaces, Connectors, Inputs, Outputs* This field is used for the systematic decomposition of the whole system. By describing relevant interfaces, connectors, inputs and outputs, the conceptual model can be refined. *Electrical connections* appearing in the readout system can be mapped to relationships between respective class candidates.

- *Timing Issues* This helps to discover time-critical tasks - a factor which is more important for the design and implementation phases than for the analysis phase.

- *Parameters Configured by APVDAQ readout application* Analysing the configuration parameters of the APVDAQ readout application helped to understand which properties of the readout system were important to the user. These could be assigned to attributes of respective class candidates. Additionally, it was possible to identify legacy parameters that are no longer needed for future purposes.

- *Data Acquired by APVDAQ readout application* All data acquired by the APVDAQ readout application from the component is analysed here. In this manner, it is possible to improve understanding about the data flows in the system and the required data structures.

- *Error States* Investigating the main error scenarios for each component is the basis for working out an exception handling strategy. For TuxDAQ, the existing source code of the APVDAQ readout application served as a reference for possible error scenarios.

- *Potential Future Modifications* The information stated in this field is particularly important for elaborations on the flexibility of the software model. For example, it should allow developers to modify the software for a newly developed board in the back-end, which comprises FADC+PROC functionality, REBO functionality and serves more readout chips than before (see Section 2.1.4).

- *Miscellaneous/Questions* This field is intended for unresolved issues concerning the component.

Based on the produced forms, small CRC cards can be created. Each CRC card represents one of the class candidates identified before and describes the *class name*, its *responsibilities* and *collaborations* with other class candidates in brief. If desired, the cards can be pinned up on a wall. With a cotton thread, they can be associated with other class candidates depending on the collaborations between respective components. This enables visually supported group discussions on the resulting class structure. Going back to forms and CRC cards, a conceptual model containing classes with attributes, operations and associations with other classes can be built up.

Using the technique presented above, it is noticeable that a corresponding class candidate exists for each type of hardware component. From an analytical point of view, this is a very efficient approach. The software design maps 1:1 to the hardware design. Therefore, the modification of a hardware module will only affect the associated software module (class). From a design perspective, the elaborated conceptual model might be considered suboptimal. Due to design or implementation issues, some class candidates may be unsuitable for the final architecture. In the concrete case of TuxDAQ development, only one single class candidate was eliminated during the design phase[48].

4.2 Object-Oriented Design

In the Object-Oriented Design (OOD), the outcome of OOA - the conceptual model - was revised with respect to software quality and the target implementation technology.

To obtain a flexible, robust, reusable and developable application, *rotting design* should be prevented. Four major symptoms [26] were used to identify a rotting design in TuxDAQ:

1. *Rigidity* Every change of the software causes a wide range of changes in dependent modules. For example, the TuxDAQ design would be rigid if the software integration of a new VME bridge resulted in a long-term task. A rigid design of TuxDAQ would cause the delay or even omission of such a minor software adaptation.

2. *Fragility* This is closely related to rigidity. If software breaks in many places every time it is changed, it is fragile. For example, with a fragile design, adaptations in the FADC+PROC software representation

[48]This was the class candidate *AlteraFPGA* representing the FPGAs of the FADC+PROC board (see Section 2.1.6). As the responsibility of this class was too small, it made no sense to implement the AlteraFPGA functionality as a dedicated class. Respective attributes and operations were implemented as members of the classes representing the FADC+PROC board.

of TuxDAQ could lead to crashes in some components with a close relation to the NECO board. With fragile software, any change in its design introduces more problems than it solves. Like rigidity, this could lead to decision makers avoiding even minor software adaptations.

3. *Immobility* This is “the inability to reuse software from other projects or from parts of the same project” [26]. In the scientific everyday work at HEPHY, there is a high turnover of student personnel. Various tasks at different levels of complexity are accomplished by student apprentices, diploma students, doctoral candidates and other staff. In such an environment, immobile software could easily lead to recurring productions of already existing code. This time could be used more efficiently for other tasks.

4. *Viscosity* When a software change method preserving the software design is harder to employ than a hack, the design viscosity is high. Also, the technical environment could make hacks more attractive to developers than design-preserving methods. Therefore, TuxDAQ should allow for design-preserving changes in a simple and time efficient manner.

Not only should these properties of poor software quality be avoided in the context of a long-term strategy for TuxDAQ, but in addition to this, preventing rotting design is vital for building up a reusable framework of components (see Requirement 3.4.3/3).

To address the aforementioned issues for TuxDAQ, some fundamental principles of OOD were obeyed. Familiarity with these principles is necessary to understand the ideas behind the TuxDAQ design. For this reason, they are described in the following sections, providing hands-on examples of the TuxDAQ concept. Then, common techniques conforming with these principles, namely *design patterns*, are discussed.

4.2.1 The Single Responsibility Principle[49]

“A CLASS SHOULD HAVE ONLY ONE REASON TO CHANGE.” [26]

In the context of this principle, a *responsibility* is defined as a *reason to change*. A class should have only one reason to change.

To understand this principle, consider the following example. As described in Chapter 2, the VME boards in the backend of the Belle II readout system can be controlled via built-in I^2C master units. To access these, specific commands have to reach the respective boards via the VMEbus interface.

At a first glance, it is reasonable to design a dedicated class representing a VME board in software. That is, it encapsulates all configuration parameters, settings, and capabilities in the scope of the VME board. Fig. 4.3a shows such a class, namely *NECO VME board*. If this class is investigated more closely, it can be noticed that it has two different responsibilities:

1. The first one is to *represent the NECO hardware component in the software*, depicted by *GetDescription()*. For instance, the GUI could provide a system explorer, which renders all hardware components of the readout system graphically. Additionally, string descriptions of each component could be provided for each component, which are loaded from a configuration file. Such a string description would be one of the attributes of the NECO VME Board class.

[49] [26]

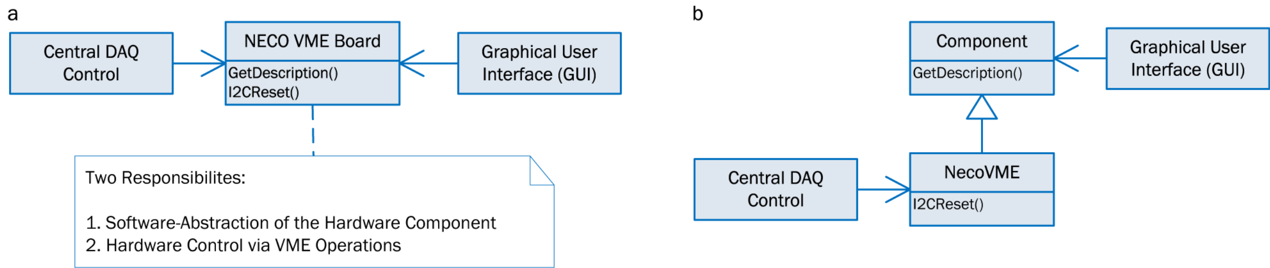


Figure 4.3: *Single Responsibility Principle (SRP): (a) Violation of the Principle, (b) Possible Solution (UML Class Diagram)*

2. The second responsibility is *NECO hardware control*, depicted by *I2CReset()*. A central DAQ control component could make use of this operation.

Obviously, the class of Fig. 4.3a violates the Single Responsibility Principle (SRP). In so doing, it puts the robustness of the software application at risk. If there is a change in the hardware control code (Responsibility 2) of the class, this is likely to cause the breaking of code related to Responsibility 1. This is due to the fact that both responsibilities are part of the same class. In this case, both the central DAQ control and the GUI would be affected by possible software errors in the class NECO VME Board.

To improve the situation, NECO VME Board should therefore be split up into two separate classes according to their responsibilities. Fig. 4.3b presents one possible solution. In this scenario, all operations representing the hardware component in the software are separated from class NECO VME Board and included in a new class named *Component*. Another class, namely *NecoVME*, comprises hardware control functionality.

This separation makes the design more robust, as any software errors due to changes in the hardware control operations no longer affect the operations related to Responsibility 1 and, therefore, the GUI of the system.

The solution presented here is one out of various options. For example, classes could be split by means of a (horizontal) association relationship rather than a (vertical) generalisation relationship. However, in some cases it would only be possible to obey the SRP with inadequate effort. A coping strategy is presented in [26].

4.2.2 The Open Closed Principle[50]

"SOFTWARE ENTITIES SHOULD BE OPEN FOR EXTENSION, BUT CLOSED FOR MODIFICATION." [26]

This principle states that modules should be designed in a way that allows for extensions without requiring the modules to change. This goal can be achieved by abstraction as illustrated in the example below:

The class *NecoVME* of Fig. 4.3b is responsible for hardware control over the NECO back-end board of the readout system. As this board is accessed via the VMEbus interface, the class will invoke several functions of the respective VME implementation, e.g. the *CAEN VME implementation* (see Fig. 4.4a).

If the CAEN VME implemenation was replaced with a new VME implementation at some point, the NecoVME class would have to be changed accordingly. This could easily lead to errors.

Fig. 4.4b gives a solution to this problem. In this design, NecoVME depends only on an *interface*[51],

[50] [26], for the primary source from the same author see [28].

[51] In C++, interfaces are implemented as *abstract classes* containing pure virtual functions only. Such an abstract class cannot be instantiated and "provides a base class from which other classes can inherit" [29].

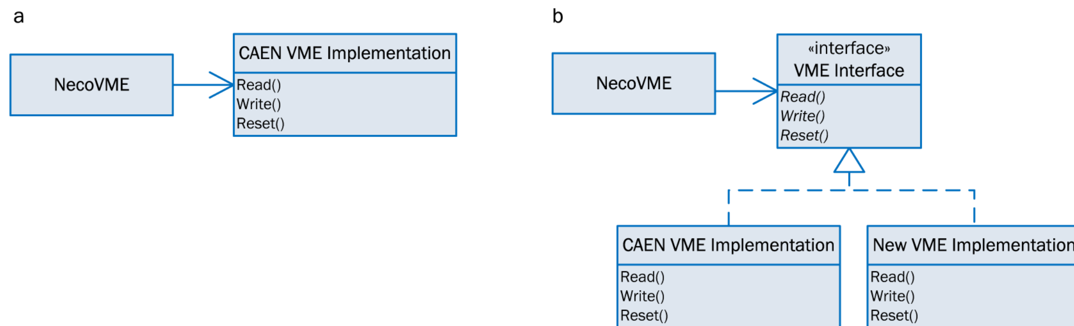


Figure 4.4: *Open Closed Principle (OCP): (a) Violation of the Principle, (b) Possible Solution (UML Class Diagram)*

namely the *VME Interface*. The CAEN VME implementation, like any *new VME implementation*, realises this interface[52]. However, NecoVME no longer depends on concrete implementations and therefore does not have to be changed when a new VME implementation arises. It has been *closed for modification*. Nevertheless, it is *open for extension* by new VME implementations.

The technique presented is only one possibility to conform to the Open Closed Principle (OCP). While this principle cannot be obeyed fully for a whole software structure, even partial OCP compliance improves software quality.

4.2.3 The Liskov Substitution Principle[53]

"SUBTYPES MUST BE SUBSTITUTABLE FOR THEIR BASE TYPES." [26]

This principle ensures that a client of a class can continue to function properly if that class is substituted by a derived class. This can only be achieved if special care is taken in regards to *derived methods*.

In brief, a base class can be substituted by a derived class, if any derived methods "*expect no more* and *provide no less*" [30]. This is the case if:

1. Its *preconditions are no stronger* than those of the base class method

2. Its *postconditions are no weaker* than those of the base class method

In terms of C++, "functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it." [30].

To reach a clear understanding of this, Fig. 4.4b should be reconsidered. For OCP compliance, the *VME Interface* was introduced. It specifies an operation *Reset()*, which is intended for a VME system reset. As it is implemented in the *CAEN VME implementation*, this operation could be used by a client.

In fact, the VME boards of the prototype system at HEPHY disregard the system reset line specified in the VMEbus standard. They can be reset by other means. Therefore, performing a VME system reset does not have any impact on the system operation. It follows that the *NecoVME* client class never invokes the Reset()

[52]These VME implementations can be considered as adapters to the manufacturer specific APIs.
[53] [26], for the primary source from the same author see [30].

operation in reality. Nonetheless, specifying such an operation makes the VME Interface reusable for other projects and future purposes.

If a new VME bridge is acquired, another, *new VME implementation* may be required (as it might be a non-CAEN product). Considering the aforementioned facts, the developer of this new class might omit the implementation of Reset(). For example, they could leave its method body blank.

This solution works well as long as the client is NecoVME. If another client uses the VME Interface, e.g. in the context of another project with VME boards taking the VMEbus reset line into account, this client might rely on Reset() to work properly. While the CAEN VME implementation would work as expected (it provides a well-functioning Reset() operation), the new VME implementation, only providing a Reset() mock-up, would break.

In this simple, hands-on example, the derived new VME implementation provides less than specified by the base VME Interface[54]. In practice, aiming at Liskov Substitution Principle (LSP) compliance requires careful examination of the software design. However, violations of this principle may lead to filthy constructs harming software quality.

4.2.4 The Dependency Inversion Principle[55]

"HIGH-LEVEL MODULES SHOULD NOT DEPEND UPON LOW-LEVEL MODULES. BOTH SHOULD DEPEND ON ABSTRACTIONS. ABSTRACTIONS SHOULD NOT DEPEND ON DETAILS. DETAILS SHOULD DEPEND ON ABSTRACTIONS." [26]

According to this principle, clients should depend on interfaces (i.e. abstract classes and abstract methods) rather than concrete classes or methods.

In procedural architectures, a high level module implements a *high level policy*. In so doing, it depends on lower level modules, which depend on yet lower level modules, and so on. Fig. 4.5a illustrates this. For example, the `main()`-function of a C-based program could rely on a VME library which, in turn, depends on operating system functions.

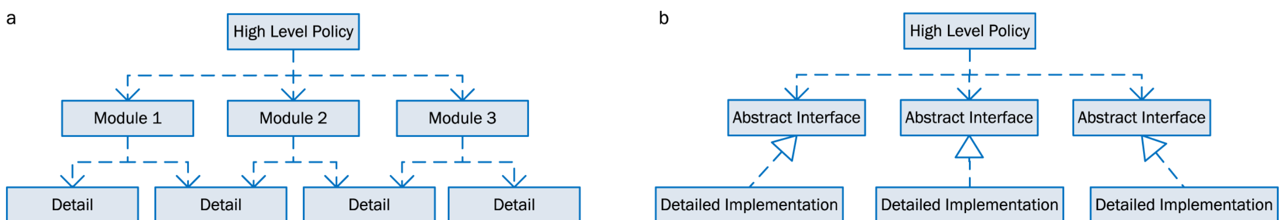


Figure 4.5: *(a) Dependency Structure of a Procedural Architecture, (b) Object-Oriented Architecture with Inverted Dependencies, See [27] (UML Class Diagram)*

Object-oriented architectures conforming to the Dependency Inversion Principle (DIP), by contrast, exhibit high level modules, which depend on (abstract) interfaces rather than concrete implementations (see Fig. 4.5b).

[54] In more formal words, the postcondition of the new, derived VME implementation is weaker than that of the base class implementation.

[55] [26], for the primary source from the same author see [31].

Moreover, detailed implementations realise these interfaces, which is why they are no longer depended upon but *depend themselves upon abstractions*.

The *dependency inversion* shown prevents clients from depending on *volatile* modules. In other words, abstract interfaces change much less frequently than concrete classes. Furthermore, high level modules are not affected as long as the interface conditions are maintained if detailed implementations should be extended.

The TuxDAQ-specific example illustrated by Fig. 4.4b not only conforms with the OCP, but it is also DIP compliant. By introducing an interface at this point, both principles are obeyed. However, following this strategy for an entire software architecture may become uneconomic, as it could be cost intensive. Therefore, it is reasonable to balance the advantages of obeying this principle against development effort in every specific case.

4.2.5 Package Architecture Design Principles

The OOD principles presented in Sections 4.2.1 to 4.2.4 address the object-oriented software design on a class level. Above this level, classes have to be grouped according to the extraction of reusable software components. Section 5.2 describes how this was achieved for TuxDAQ. The underlying principles also deserve closer attention and are outlined in [26].

4.2.6 Design Patterns

Following the design principles described above consistently requires engineers to have wide experience in OOD. It can be seen that recurring structures appear at different places. Such recurring structures are also referred to as *design patterns* [27].

In 1995, Gamma et al. published a substantial book [32] on common design patterns, which is sometimes referred to as the *Gang of Four (GoF) Book*[56]. The contributions related to the object-oriented world are now seen as fundamental. As per the GoF definition, a design pattern systematically "names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design". Among other information, each design pattern comes with a problem description, the solution, possible applications, and its consequences. Design patterns are reusable *best practice* techniques that *obey the principles of good software design*.

Due to the fact that software developers in the TuxDAQ environment are typically physicists or electrical engineers with limited experience in OOD, the application of design patterns is a good way to *achieve* and *maintain* a high level of software quality with reasonable effort. Development time can be minimised, as sophisticated solutions are provided for complex problems. Among engineers who know the major design patterns, a higher level of software design can be achieved. For example, just naming a design pattern may save time-consuming explanations.

Design patterns can have either an *architectural*, *creational*, *structural* or *behavioural* purpose. Architectural patterns describe a system organisation that has proven successful in previous systems [20]. Creational patterns are concerned with the creation of objects. Structural patterns define how classes or objects are composed. Behavioural patterns describe the interaction between classes or objects and how responsibilities are distributed among them [32].

[56] The book was written by four authors.

In the architecture of TuxDAQ, the following patterns[57] can be found:

- *Architectural:* Model-View-Controller (MVC)
- *Creational:* Abstract Factory
- *Structural:* Adapter, Composite
- *Behavioural:* Iterator, Observer, Strategy

In Chapter 5, they are discussed based on concrete applications in the TuxDAQ design and implementation.

4.3 Object-Oriented Programming

The OOD principles and techniques presented above aim at a software design that does not rot in the course of time. This will make future maintenance and extension easier and faster to achieve.

A strategy for Object-Oriented Programming (OOP) should work towards a program that is easy to read, test and debug, and can be adapted to changing requirements. Moreover, the program should not behave strangely at any point [29]. If these conditions are met, reusability, maintainability and extendibility cannot only be achieved for the design, but also for the implementation.

There are numerous *software engineering principles* that can be followed to produce good C++ code. An exhaustive compilation of these principles would be beyond the scope of this thesis, but is provided by [29]. However, in the course of TuxDAQ programming, some basic *restrictive and error-preventive measures* were taken. These are presented in the following sections.

4.3.1 Relying on Stable Software Only

In terms of software, the word *stability* may refer to two different things:

1. *Runtime stability of an application* In daily use, a software application is said to be stable if its crashing during operation is not likely.
2. *Low frequency of changes of a package or system configuration* Whether in the context of software design or operating systems, software packages are referred to as stable if they change infrequently.

The aim of the TuxDAQ implementation strategy was to achieve both types of stability; only stable software was used for software development. This refers to development tools, the operating environment and any third party standard components (frameworks and toolkits). Obtaining both types of stability both minimises the software development time as stability issues can be minimised and increases the stability of TuxDAQ.

Following this principle, it was decided to comply with the established C++ standard *ISO/IEC 14882:2003*. Although work on a new standard began in 2003 [29] and some compilers already support many of its features, neither the standard nor respective compilers could be considered stable when the TuxDAQ project began.

[57] MVC: [33], other patterns: [32]

4.3.2 Following Coding Conventions

For a software project size of TuxDAQ, following standardised coding conventions yields some advantages:

- Generally, program code becomes clear, organised and aesthetic. This improves readability and makes the program easier to understand. As a consequence, preventing and finding errors becomes easier.
- In a project environment like HEPHY, with several engineers involved, it is important to enable fast, smooth handover of the software project from one developer to another. This can only be put into practice if the program code is based on common coding conventions, enabling succeeding programmers to become familiar with existing code faster due to improved readability.
- Even if several developers were involved in the development of one program, coding conventions would contribute to ensuring a consistent quality for a continuance.

As the CERN ROOT framework is popular in the environment of HEPHY (see Section 5.3), it is likely that succeeding software developers will be familiar with it. For this reason, it has been decided to follow the *C++ coding conventions of CERN ROOT* [34]. Having said that, some of the rules have been modified:

- *Prefixes and suffixes* Any suggested prefixes to class names as well as all kinds of suffixes to all types of names were omitted. Instead, the prefixes *p* for pointers and *r* for references were deemed reasonable.
- *Comments and source file layout* The CERN ROOT framework comes with a documentation tool for generating web-based documentation out of properly documented source code. Due to a lack of stability of this tool, it was decided to choose another appropriate program, namely Doxygen (see Section 5.1). As a consequence, the CERN ROOT conventions concerning comments were neglected. Instead, TuxDAQ code provides Doxygen compliant comments. For the source file layout, all conventions regarding the commenting issue and CERN-specific version control were ignored.

4.3.3 Focusing on Clarity

TuxDAQ is a software application with a close relation to sophisticated readout hardware. Seemingly small software errors may cause wide-ranging troubleshooting in hardware. To minimise such software errors, it is important to focus on *clarity* of program code. The clearer software code is written, the fewer potential programming errors will arise.

There are several strategies to achieve clarity. For TuxDAQ programming, the most important ones were:

- *Using meaningful names* All components of the software code (variables, constants, classes, etc.) should be given expressive names. This creates a self-descriptive program that is easier to understand.
- *Avoiding confusing, complicated code constructs* The more complicated a code construct is, the more prone it is to errors. Therefore, *keeping it simple* is a good strategy. It could be assumed that a specific complicated construct is necessary for performance reasons. However, modern compilers are capable of sophisticated performance optimisations and it makes sense not to “second-guess the compiler” [29].

- *Creating comprehensive inline documentation* At first glance, creating comprehensive inline documentation involves a significant temporal burden. But in the long run, this documentation enables all developers involved to understand the program code and by this means reduces development time and effort.

- *Using parentheses in multi-part expressions* Many logical software errors can be traced back to wrong assessment of operator precedence. For this reason, and to improve readability, using parentheses establishes clarity.

- *Choosing explicit approaches over implicit ones* Explicit approaches often require more lines of code than implicit ones. Nevertheless, in most cases, they are easier to understand. Therefore, implicit constructs like implicit type conversions, the implicit use of conversion constructors, etc. should be avoided if possible.

4.3.4 Using the Preprocessor and Templates with Care

Many preprocessor features are more appropriate for C programmers than for C++ programmers [29]. Consequently, the preprocessor was utilised as little as possible within the TuxDAQ program code. That is, it was mainly used to *include header files* and for *preprocessor wrappers* within header files to prevent duplicate declarations.

C++ provides so-called *templates* as a technique of *generic programming*. However, an extensive use of templates may result in confusing program code that is hard to understand. To prevent this, templates were only introduced when deemed clear and supportive[58].

4.3.5 Drawing Up an Error Treatment Strategy

During the runtime of an application, a situation could arise where failures produce undesired effects like memory leaks, resource leaks, corrupted data or invalid output. To prevent these and similar effects, it makes sense to draw up a *deliberated error treatment strategy* ahead of programming. This is especially true for C++, as this versatile programming language demands more cautious ways of proceeding than others do. If no carefully thought out error treatment strategy exists, the software implementation might dissolve into chaos over time.

Specific care needs to be taken regarding the exception handling mechanism of C++. Otherwise, the aforementioned effects could arise easily in the presence of exceptions. Therefore, the error treatment strategy should aim at *exception-safe code*.

An operation is said to be exception-safe, if the object remains in a *valid state* although the operation has been interrupted due to an exception. This valid state might be an error state which has to be cleared up, but the state has to be well-defined [35].

In this context, an operation of an object can give different *exception guarantees* to its clients [35]:

1. *The basic guarantee* If the operation throws an exception, the invariants of the object are preserved and no resources are leaked.

[58]This strategy is also recommended by the C++ coding conventions of CERN ROOT (see Section 4.3.2).

2. *The strong guarantee* If the operation throws an exception, it leaves the program state exactly as it was before the operation was invoked.

3. *The no-throw guarantee* The operation will not throw any exception.

For TuxDAQ, the following error treatment strategy was worked out:

- "Do not use exception handling as an alternate form of flow control. These additional exceptions can get in the way of genuine error-type exceptions" [29].

- Create functions with common error conditions that return an error code rather than throw exceptions [29].

- Consider for every operation, which exception guarantee it gives in the presence of an exception.

- Write exception-safe code. For this purpose, using the programming idiom of *Resource Acquisition Is Initialisation (RAII)* [35] is vital.

- Implement the *terminate()* function to prevent resource leaks [29].

- Document all exceptions an operation can throw and catch them unfailingly at a well-considered place in the program.

4.3.6 Portability Issues

Platform variations (e.g. different operating systems, compilers, etc.) can make portability of an application difficult to achieve. For example, the APVDAQ readout application was compiled and executed on a 32-bit Windows platform. In contrast, TuxDAQ runs on a 64-bit Linux platform. Data widths of fundamental types like integers are different on both platforms. Referring to Requirement 3.4.2/3, both programs are expected to produce the same data output files. If this issue is not dealt with sufficiently, data alignment may differ in both programs.

Portability is elusive, but some simple measures can be taken to make it easier:

- *Avoid raw C types* Following the C++ coding conventions of CERN ROOT, the use of raw C types like *int* was avoided. Instead, abstract data types as provided by the CERN ROOT framework [36], e.g. *Int_t*, were used. The type definitions for these data types are constructed in a way that guarantees unique data widths across platforms.

- *Avoid system-specific calls* Whenever possible, platform neutral calls were used instead of platform-specific calls.

5

Software Design and Implementation

This chapter describes the software design and implementation of TuxDAQ. It summarises the basic concepts in a way that allows the reader to become familiar with the software engineering background of the program in a short time.

Any diagrams introduced in this chapter make no claim to be complete. Instead, they aim to draw attention to important issues and serve as brief introductions to concepts involved. For the sake of clarity, any namespace identifiers have been omitted when referring to class names. Compared to the original program code, some of the code fragments presented were slightly adapted and abridged to improve readability.

TuxDAQ has been implemented with C++, which is taken into account as follows:

- *No interfaces in diagrams* As per Footnote 51 (see page 37), diagrams show abstract classes instead of interfaces.

- *C++ jargon* This is preferred to the more general terminology of OOA and OOD. For example, in C++, *methods* are referred to as *member functions*. Moreover, a *generalisation relationship* between classes is implemented by means of an *inheritance relationship* in C++.

5.1 Documentation

As per Requirement 3.4.10/1, an *exhaustive web-based documentation of the TuxDAQ API* was created with the help of the Doxygen documentation system [37]. Following the syntax rules of this tool, the header files of TuxDAQ were provided together with respective source code comments, and the tool also allowed for an automated generation of the documentation based on these comments.

The API reference manual characterises classes and namespaces, class attributes, class member functions, constants, typedefs and enumerations on a detailed level. When deemed necessary, electronic parameters and constraints of the readout hardware are described. Class inheritance diagrams are presented, and any exceptions a method may throw described. Programmers should consult the API documentation as a detailed implementation reference, which is available in the TuxDAQ repository (see Appendix B).

In addition to Doxygen-specific comments in the header files, complex program artefacts have been provided with adequate inline comments[59] (see Requirement 3.4.10/2).

5.2 Reuse-Oriented Architecture

In the design phase, particular attention was paid to the following types of *software reuse*:

- *Reuse of established third party*[60] *software components*[61] To avoid ‘reinventing the wheel’, the design process aimed at reusing already existing standard components for common tasks. By this means, software development effort could be reduced together with cost and risk [20].

- *Extraction of reusable software components* Classes and dependencies between them were organised in a way that allowed for the extraction of self-contained, reusable components. This was done particularly with regard to Requirement 3.4.3/3 (Reusability for the Belle II Experiment) and reusability for TuxOA (see Section 3.2).

Fig. 5.1 shows how the resulting software architecture addresses these goals. The bottom three collections of components build up a framework reusable for the Belle II Experiment. The components above are specific to the TuxDAQ, utilising and extending the components below for this purpose.

The architecture presented here should not be confused with a *layered architecture*, where the software system is organised in several layers with each of these layers *only* relying on the facilities and services provided by the layer *immediately beneath* it [20]. This stands in contrast to the TuxDAQ architecture, where, for example, classes of a TuxDAQ-specific component utilise classes of the C++ Standard Library.

5.3 Third Party Standard Components

As described in the previous section, several standard components were used for common tasks. Selecting such components requires detailed knowledge of the implementation technology at hand, namely C++. Not only

[59] Some descriptions appearing in the API reference manual and some inline source code comments have been carried over from the APVDAQ readout application.

[60] The term ‘third party’ refers to people or organisations other than the software developers and users of TuxDAQ.

[61] “Deployable, independent units of software that are completely defined and accessed through a set of interfaces” [20]

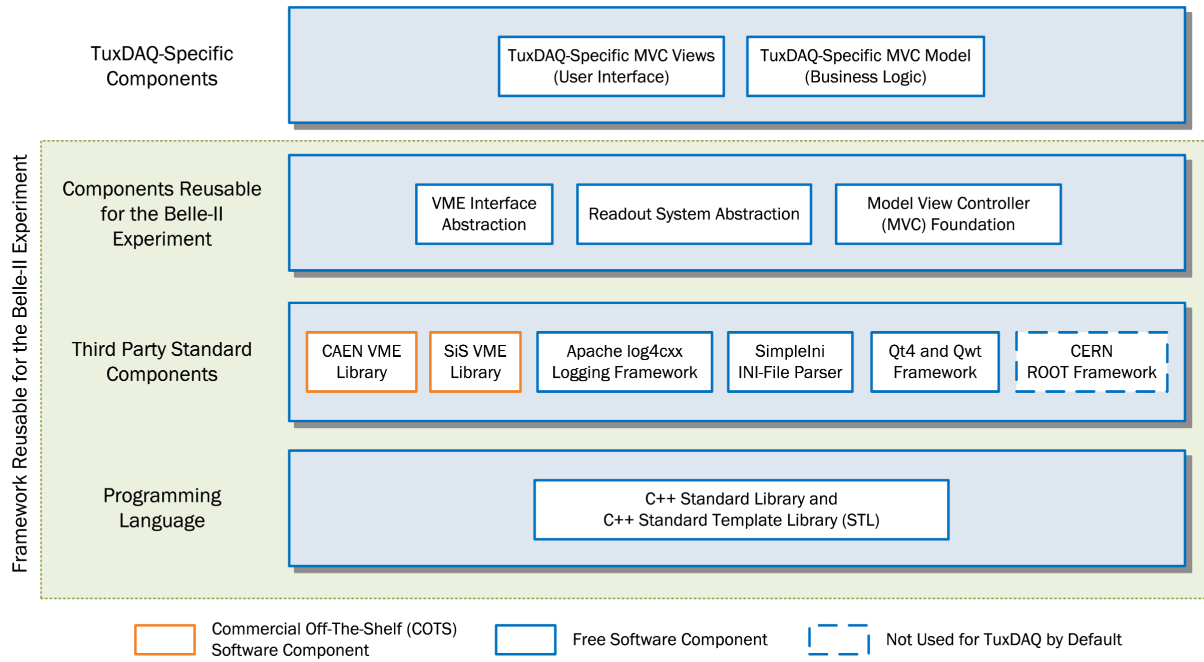


Figure 5.1: *Reuse-Oriented Architecture of TuxDAQ*

is it necessary to have a good overview of the necessary components that are available; in addition to that it is important to be able to estimate the development effort a component requires. If a component entailed an inadequately long training period, it might make more sense to find an alternative.

For this reason, it has been decided *not* to use the following:

- *The Boost C++ Libraries* [38]. Although these libraries are very common in the context of C++ and have several benefits, they would demand a substantial training effort from developers without detailed knowledge.

- *The CERN ROOT Framework* [39]. This is an open-source, free, large-scale framework for particle physics applications which provides numerous functions for related purposes. As per the original plan, TuxDAQ was supposed to be based on this framework due to the fact that it is widely spread in the related research area. However, similar to the Boost C++ Libraries, CERN ROOT requires a high effort of initial training. Additionally, this framework has shown shortcomings in robustness in personal evaluation, and its dictionary-based implementation was assessed as inappropriate for TuxDAQ purposes. As TuxDAQ does not perform any online processing and analysis on acquired data (see Requirement 3.4.5/2), the beneficial functions of CERN ROOT are not used, and the disadvantages would outweigh the advantages of using this framework. On that account, it was decided together with the HEPHY staff, not to use CERN ROOT for TuxDAQ, but for TuxOA. Nevertheless, with respect to Requirement 3.4.9/5 (compatibility with the CERN ROOT framework), TuxDAQ allows to incorporate CERN ROOT, and the TuxDAQ source code follows the coding conventions of this framework, as described in Section 4.3.2.

The selection of appropriate standard components was not only driven by the elaborations mentioned above, but also by Requirement 3.4.9/1 (TuxDAQ as open source project). To address the functional Require-

ments 3.4.2/2 (Multiple VME bridges), 3.4.5/11 (Offline run reproduction), 3.4.7/10 (Offline error tracking), 3.4.4/1 (Configurability by means of a configuration file) and 3.4.8/1 (Graphical User Interface (GUI)) with minimal development effort, the following third party components were evaluated as appropriate for TuxDAQ:

- *The CAEN and SiS VME Libraries*[62] These libraries come with the VME bridge hardware and are implemented in C. The VME libraries are Commercial Off-The-Shelf (COTS) components[63] and cannot be considered free software. This has to be taken into account when TuxDAQ is deployed in research facilities other than HEPHY. In this specific case, however, it can be assumed that appropriate, licensed libraries are available in the target system.

- *The Apache log4cxx Logging Framework* [40]. Patterned after Apache log4j [41], this C++ based logging framework allows for a flexible, performance-optimised logging on different levels and a low-tech debugging of applications. The output target(s) and format(s) can be defined in a configuration file by the user without the need to recompile software . The API provided by log4cxx is simple and does not require extensive training. Using the Apache license [42], log4cxx is open-source and free.

- *SimpleIni* [43]. This is a compact C++-based library for reading and writing INI-style configuration files in a simple way. It supports several character encodings and provides cross-platform capabilities. This is particularly helpful when legacy configuration files of the Windows-based APVDAQ readout application are to be used by the Linux-based TuxDAQ. Using the Massachusetts Institute of Technology (MIT) license [44], SimpleIni is open-source and free.

- *Qt and Qwt* [45, 46]. With Qt, a proven state-of-the-art multi-platform framework was selected for the implementation of the GUI. For developers experienced in user interface programming, the initial training effort is modest. A variety of widgets and a well-designed software architecture allowed to create a presentable and robust GUI in a decent time frame. It is important to mention that Qt components were exclusively used for the GUI subsystem of TuxDAQ. Therefore, the components of the architecture not concerned with the GUI are entirely independent from Qt. This improves the maintainability of the architecture. The Qwt library provides additional Qt widgets for engineering purposes, e.g. curve plots. Using the GNU Lesser General Public License (LGPL) [47], Qt is open-source and free. Apart from a few minor exceptions, Qwt also uses the LGPL.

5.3.1 Namespaces

All TuxDAQ program artefacts (except those belonging to third party components) have been assigned respective *namespaces*:

- *hephy::el* This namespace is intended for all components reusable for the Belle II Experiment.

- *hephy:el:tuxdaq* This namespace is dedicated to components specific to TuxDAQ and therefore not reusable.

[62] As the NI VME bridge available at HEPHY lab is outdated (see Section 2.1.7), an according software implementation was not considered high priority for TuxDAQ development.

[63] Commercial systems in their own right that provide specific functionality [20]

5.4 VME Interface Abstraction

As per Requirement 3.4.2/2, TuxDAQ is expected to operate with different kinds of VME bridges, and specifically the CAEN A2818/V2718 and SiS 1100e/3104 bridges have been considered for TuxDAQ. Both come with C-based software libraries for Linux providing appropriate APIs for programmers. Each manufacturer follows a proprietary concept of hardware access, error types and resource acquisition.

To be portable across different VME bridges, TuxDAQ should not hard-code any classes for a particular VME bridge. Invoking manufacturer-specific functions throughout the application would make it difficult to change the VME bridge in the future. Therefore, the solution had to incorporate the requirements below:

- Keep TuxDAQ-code widely independent from the way how VME-specific classes are created, composed and represented.
- Allow for a configuration of TuxDAQ with one of multiple VME bridges with reasonable expense.
- Allow succeeding software developers to tie in new VME bridges in a short period of time.

Responding to this challenge, the *Abstract Factory* [32] design pattern was followed for implementation. Fig. 5.2 gives an overview of relevant classes and the relationships among them.

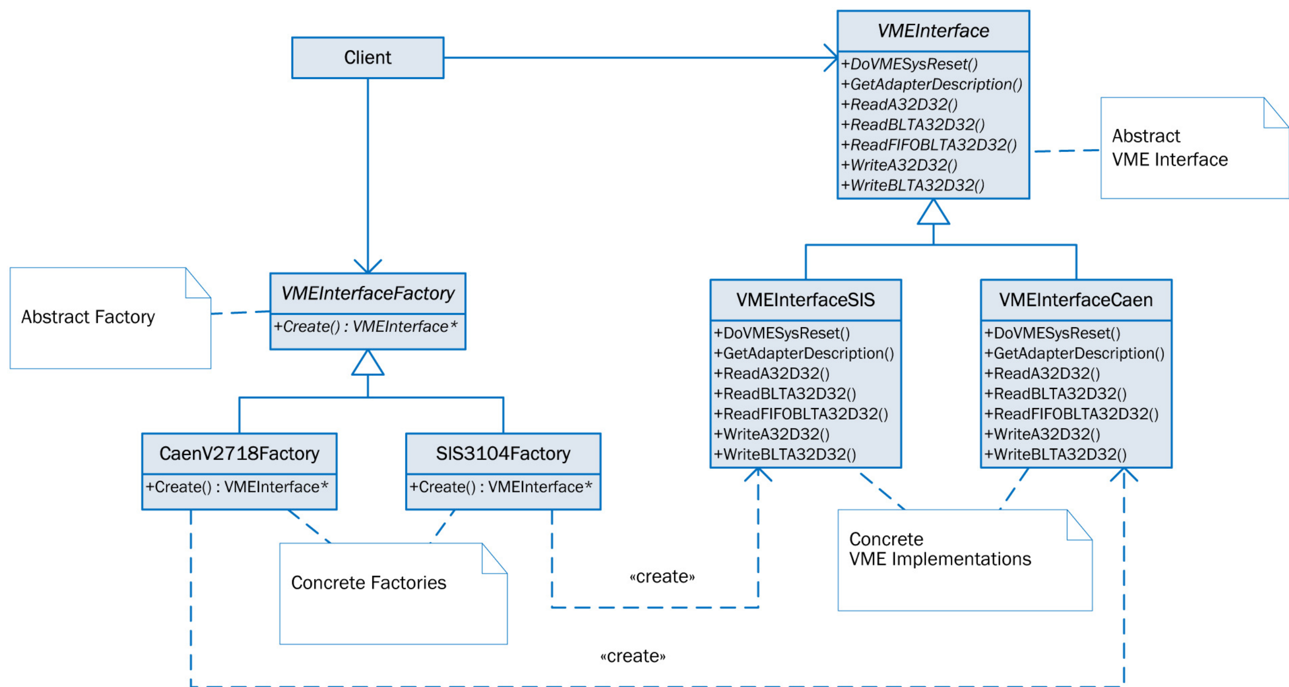


Figure 5.2: *VME Interface Abstraction: Abstract Factory Pattern (UML Class Diagram)*

The abstract class *VMEInterface* defines a unified interface for VME access, which is used by all clients[64]. *VMEInterfaceSIS* and *VMEInterfaceCaen* implement this interface, as they are derived from VMEInterface. The member functions they provide are *wrapper functions* adapting manufacturer-specific APIs to VMEInter-

[64]In this context, any TuxDAQ class using this interface is referred to as client.

face[65]. These member functions allow the performing of various VME read and write operations based on different address and data widths, retrieving information about the connected VME controller and performing a VME system reset.

Like all classes of TuxDAQ, VMEInterfaceSIS and VMEInterfaceCaen follow the idiom of *RAII* [35]. With RAII, resource management is tied to the lifespan of an object. Resources are allocated when an object is created, and deallocated when it is destroyed. In so doing, exception safety can be accomplished. For VMEInterfaceSIS and VMEInterfaceCaen, this means that a specific VME bridge is opened during initialisation of the corresponding object. There is no chance to use this object prior to its constructor having been completed. Consequently, the object cannot be used unless the VME bridge has been opened properly. If the initialisation of the VME bridge fails, an exception is thrown, and the object is not created. Similarly, the VME bridge is released by the destructor of such an object. In C++, the only code definitely executed after an exception is thrown, are the destructors of objects residing on the stack. In other words, the deallocation of the VME bridge is guaranteed to happen, even if an exception has occurred. By following RAII, it is guaranteed that, for example, a VMEInterfaceCaen object only exists and remains in a *consistent state*, if the underlying VME bridge has been acquired properly. Additionally, RAII *prevents resource leaks*. As an example, Listing 5.1 shows both the constructor and destructor of VMEInterfaceCaen.

Listing 5.1: *Following the RAII Idiom: Constructor and Destructor of VMEInterfaceCaen*

```
1  VMEInterfaceCaen::VMEInterfaceCaen(const CVBoardTypes boardType, const Short_t linkNr,
       const Short_t boardNr) :
2      fVMEHandle(VMEInterfaceCaen::kNULL_HANDLE), fVMEBoardType(boardType)
3  {
4     const CVBoardTypes& rBoardType = this->GetVMEBoardType();
5     const Int_t& rVME = this->GetVMEHandle();
6
7     // Initialise CAEN VME device:
8     CVErrorCodes errCode = CAENVME_Init(rBoardType, linkNr, boardNr, &rVME);
9     if (errCode != cvSuccess) {  /* If error occurred, throw exception */ }
10 }
11
12 VMEInterfaceCaen::~VMEInterfaceCaen()
13 {
14    const Int_t& rVME = this->GetVMEHandle();
15
16    CVErrorCodes errCode = CAENVME_End(rVME);   // Deinitialise CAEN VME device
17    if (errCode != cvSuccess) { /* If error occurred, output an error message. */ }
18 }
```

According to line 8 of the listing, the CAEN API requires a board type, a link number and a board number to identify the actual VME bridge. This determines the constructor of VMEInterfaceCAEN (see line 1) and, therefore, the way respective objects are instantiated. In contrast, the API of SiS uses Linux device handles and a ‘minor device identifier’ to address a specific VME bridge. As a result, VMEInterfaceSIS provides a constructor different to the one of VMEInterfaceCAEN.

[65]If VME manufacturers provided classes rather than non-object-oriented function libraries, VMEInterfaceSIS and VMEInterfaceCaen could be classes based on the *Adapter (Wrapper)* design pattern [32]. However, given that the libraries at hand provide C-functions only, VMEInterfaceSIS and VMEInterfaceCaen are mere classes providing wrapper functions.

To enable TuxDAQ to be widely independent from manufacturer-specific details, it is necessary to abstract the way VME adapters are represented as well as the way of object creation. This is achieved by factory classes, each of which is intended to create a specific type of 'product' (VME implementation). That is, the class *CaenV2718Factory* provides the factory function *Create()* for the creation of a VMEInterfaceCaen instance in the *free store*. In contrast, the Create() function of *SIS3104Factory* creates an object of type VMEInterfaceSIS in the free store (see Fig. 5.2 for the class diagram).

The factory functions *do not take* any arguments. Instead, they hard-code the arguments required by the specific constructors of VMEInterfaceCaen and VMEInterfaceSIS[66]. The abstract class *VMEInterfaceFactory* ensures that all derived classes, namely *CaenV2718Factory* and *SIS3104Factory*, implement this function with the same (parameter-less) signature. By this means, a client can request the creation of different types of objects from different factory classes with the same function call, namely Create()[67].

Although the factory functions create concrete objects, either of type VMEInterfaceCaen or VMEInterfaceSIS, they return pointers of the base type VMEInterface*[68]. Therefore, a client may invoke the Create() function of CaenV2718Factory to instantiate a concrete VMEInterfaceCaen object, but only gets a pointer of the abstract base type VMEInterface* to this object. Due to the 'is a' nature of the generalisation relationship, the VMEInterfaceCaen object *is a* VMEInterface object as well. This enables the invoking of all the VMEInterface-specific member functions of the VMEInterface object via a VMEInterface pointer. With this design, the only dependency of the client on a specific VME bridge lies in the selection of the appropriate factory class. The client is not aware of any concrete VME implementation (e.g. VMEInterfaceCaen).

In TuxDAQ, the selection of the appropriate factory class is executed in the program's main() function (see Listing 5.2). It can be seen that a pointer to a concrete factory is passed to the constructor of *TuxDController* (Lines 9, 12, 16). In this scenario, TuxDController represents the client. Listing 5.3 shows the signature of its constructor. Obviously, it takes an VMEInterfaceFactory* argument, which is again an abstract base type instead of a concrete implementation type. In so doing, the client becomes independent from both concrete products (i.e. VME interface implementations) and concrete factories. The concrete factories get instantiated at a single point - specifically at the top of main(). From there, a base type pointer is passed to the client, which can call the factory's Create() function - irrespective of whether the object is of type CaenV2718Factory or SIS3104Factory. As in the scenario described above, the client is not aware of the concrete object type behind the pointer. Nevertheless, Create() will be invoked on a concrete factory object, and it will create another object representing a concrete VME implementation.

Following the Abstract Factory pattern, all TuxDAQ code outside main() depends on abstractions only. If the VME bridge is to be changed, only the code in main() has to be modified, the rest of the program remains the same. This improves the software quality of TuxDAQ as described in Section 4.2.

It should be mentioned that introducing the abstract classes VMEInterface and VMEInterfaceFactory ensures compliance with both the Open Closed Principle (OCP) and the Dependency Inversion Principle (DIP) (see Section 4.2).

---

[66] As there is only one specific VME bridge used at one time for accessing the prototype readout system, VME bridge parameters never change. This allows hard-coding the constructor arguments of respective objects.

[67] This feature is referred to as *polymorphism*.

[68] As they derive from this class, VMEInterfaceCaen and VMEInterfaceSIS are implicitly of type VMEInterface.

Listing 5.2: *main() Function as Single Point of Concrete Factory Instantiation*

```
1  int main(int argc, char* argv[])
2  {
3     // ...
4     SIS3104Factory vmeFactorySIS;
5     CaenV2718Factory vmeFactoryCAEN;
6     // ...
7     // Select appropriate factory for VME interface (SIS/CAEN):
8     if (2 == argc && std::string(argv[1]) == "sis") {
9        TuxDController controller(&vmeFactorySIS);
10       controller.Run();
11    } else if (2 == argc && std::string(argv[1]) == "caen") {
12       TuxDController controller(&vmeFactoryCAEN);
13       controller.Run();
14    } else {
15       // default: CAEN
16       TuxDController controller(&vmeFactoryCAEN);
17       controller.Run();
18    }
19    // ...
20 }
```

Listing 5.3: *Signature of the TuxDController Constructor*

```
1     TuxDController(VMEInterfaceFactory* const pFactory);
```

5.4.1 Exception Handling

Different VME bridges do not only come with various types of resource acquisition, but they also provide different types of errors that might occur. It is desirable to achieve independence of TuxDAQ from manufacturer-specific details in this respect.

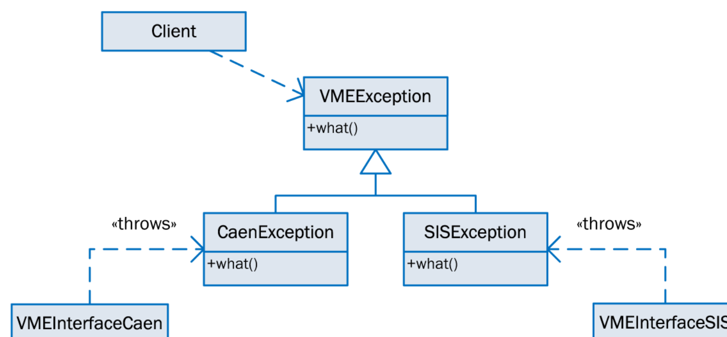


Figure 5.3: *VME Exception Hierarchy (UML Class Diagram)*

For that purpose, the exception class hierarchy presented in Fig. 5.3 has been developed. The client depends on the class *VMEException* only, i.e. it will exclusively catch exceptions of this base type. The class provides a *virtual* member function *what()*, which returns a string representation of a generic VME error type. In addition to that, the classes *CaenException* and *SISException* are intended for manufacturer-specific error messages.

Both of these classes provide a derived virtual member function what(), which adds a specific VME error string to the generic VME error message produced by the base function VMEInterface::what().

Concrete VME implementations like VMEInterfaceCaen throw specific exceptions like CaenException. The client, however, is not aware of the concrete exception type that was thrown, but retrieves a VMEException reference to it. Nevertheless, when the client invokes what(), the derived virtual member function is executed, which returns a specific error message.

5.4.2 Tying in a New VME Bridge

To tie in a new VME bridge, the following measures have to be taken:

- Create a concrete VME implementation class, which inherits from VMEInterface.
- In an error scenario, this class throws a manufacturer-specific exception. Create an adequate exception class which inherits from VMEException.
- Create an appropriate factory class, which inherits from VMEInterfaceFactory.
- Adapt the code of main() accordingly.

Note that tying in a new VME bridge does not lead to any modifications of existing code outside main().

5.5 Readout System Abstraction

The Belle II SVD readout system is a chain of various components. As already mentioned in Section 4.1, it makes sense for every type of hardware component to be represented by a dedicated *class* in software. Consequently, each component of the readout chain will be represented by an *object* that encapsulates all data and behaviour associated with the component. Relationships among hardware components will correspond to respective object relationships.

Considering the readout system more closely, it can be seen that some components are composed of other components, e.g. a detector module is composed of multiple APV25 readout chips. Other components, like NECO, are self-contained.

A simple implementation could introduce classes for primitive components (e.g. APV25) and other classes representing containers (e.g. detector modules) for these components. Following this approach, however, creates a problem: In the readout system, the containers are components at the same time, each containing all attributes and behaviours of a primitive component. For example, in the same way as APV25 objects have a unique component number, detector modules have such a number to identify them within the hardware setup. If a client intends to output these numbers for all components, there is no need to distinguish between primitive components and containers. In this scenario, the client is only interested in an attribute that is common to all components. In the simple implementation approach mentioned above, however, the client must treat primitive and container objects differently, even if the use case treats them identically, i.e. the same type of number is output for all objects.

The *Composite* [32] design pattern addresses these aspects. It “composes objects into tree structures to represent part-whole hierarchies” and “lets clients treat individual objects and compositions of objects uniformly”.

As shown in Fig. 5.4, the readout system abstraction of TuxDAQ is based on this pattern[69]. The class *Component* represents *both* primitive components and containers. It defines the data members and member functions that are common to all components. *Composite* represents containers only. This class has an *aggregation* relationship to Component, which is typical for containers. In the context of this relationship, a Composite object is referred to as the *parent* object, and the Component objects are called *child* objects. In addition to the aggregation relationship, Composite is *derived* from Component, which makes every Composite a Component, too. This constellation forms the heart of the design pattern: Every Composite can *contain* several Components, and at the same time it *is* a Component that can be treated as such.

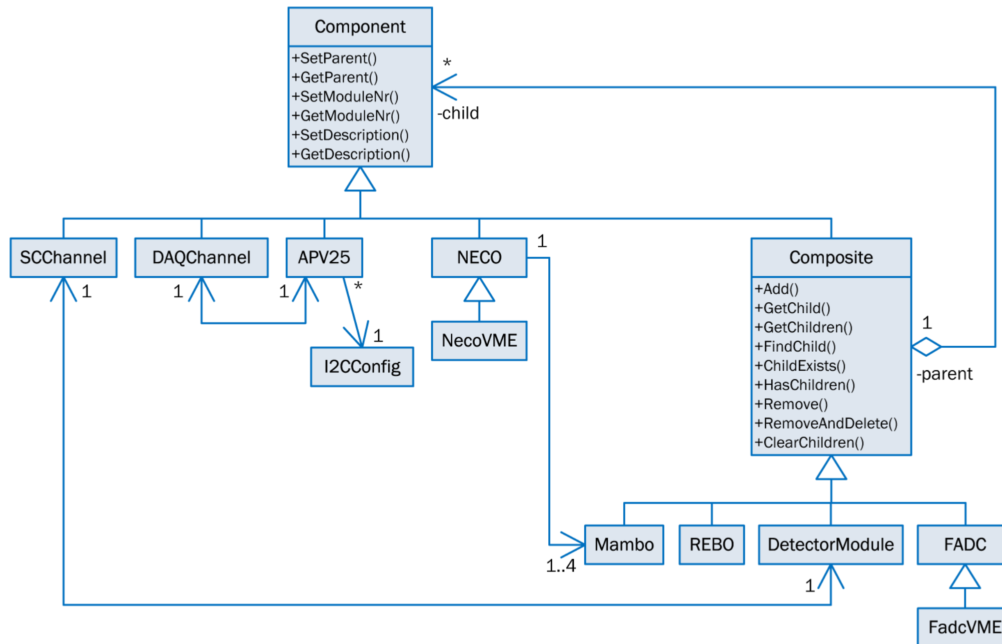


Figure 5.4: *Readout System Abstraction: Composite Pattern (UML Class Diagram)*

Both classes Component and Composite provide member functions to define and dissolve a parent-child relationship between each other. Moreover, Composite provides various functions for accessing and managing its children.

Classes representing primitive components (*SCChannel*[70], *DAQChannel*[71], *APV25*, *NECO*) are derived *directly* from Component. These are referred to as *leaf classes*. In contrast, classes representing container components (*Mambo*, *REBO*, *DetectorModule*, *FADC*) are derived from Composite[72].

With this design, the following can be achieved [32]:

- Even though a client might expect a primitive object, it can also take a composite object.
- Clients can treat containers and primitives the same.
- It is easy to add new kinds of components. Newly introduced leaf classes or subclasses of Composite automatically work in the context of existing client code without the need to modify existing components.

[69]The solution presented here is slightly different from the one provided by [32].
[70]SCChannel is an acronym for *Slow Control (SC) Channel*.
[71]DAQChannel is an acronym for *Data Acquisition (DAQ) Channel*.
[72]Due to transitivity of the generalisation (and inheritance) relationship, these classes are implicitly of type Component, too.

The classes involved have the following responsibilities:

- APV25, NECO, Mambo, REBO, DetectorModule and FADC encapsulate all data and behaviour of associated hardware components.
- The APV25 readout chips require settings related to their I^2C bus interface. Most chips use the same settings, but in some cases, they can vary across different chips. The APV25-related I^2C settings are encapsulated by class *I2CConfig*. As shown in the class diagram, one APV25 object can only be associated with one particular I2CConfig object, but an I2CConfig object can be associated with several APV25 objects.
- According to the Single Responsibility Principle (SRP) (see Section 4.2.1), the hardware components NECO and FADC+PROC are represented by *two* classes *each*. The classes NECO and FADC encapsulate all data and behaviour of the particular hardware components except their VME-based functionality. This is covered by the classes *NecoVME* and *FadcVME*, which are derived from NECO and FADC. As a result, each of the classes involved has only one responsibility, and therefore only one reason to change. As a consequence, classes NECO and FADC become reusable for TuxOA, which does not require any VME operations. Moreover, changes in the VME subsystem will not have any impact on the classes NECO and FADC. This improves the robustness of client code that is not concerned to VME-based functionality.
- The slow control path (SC channel) between a REBO board and a detector module is represented by the class SCChannel. This class encapsulates all data and behaviour associated with such an SC channel. Similarly, the class DAQChannel represents the readout path (DAQ channel) between a respective APV25 chip and the associated readout channel of an FADC+PROC board.

As per Requirement 3.4.4/1 (Configurability by means of a configuration file), TuxDAQ allows the hardware structure of the readout system and several parameters for the configuration of its components to be defined in a configuration file. This configuration file is loaded either manually or automatically at runtime (see Requirement 3.4.5/5 (Typical workflow)).

Configuration file parsing and the associated error treatment are achieved by a class called *DAQControl*, which is described in Section 5.6.1. This class creates instances of the classes described above and associates them with other objects according to a particular configuration file. Fig. 5.5 provides an object diagram for a typical configuration at the HEPHY lab, representing the following hardware setup:

- *Three detector modules* One detector module is attached to the n-side of an SSSD with 512 strips. It consists of four APV25 readout chips. The other two detector modules are attached to the n- and p-side of a DSSD, each of which contains 128 strips. Both of these also contain one APV25 chip for the readout of one respective sensor side.
- *Two REBO boards* The detector modules attached to the n-sides of sensors are served by one REBO board, the p-side detector modules by another REBO board. Three detector modules require three dedicated SC channels.
- *Two FADC+PROC boards* The readout of the sensor n-sides is accomplished by one FADC+PROC board, the p-side readout by another one. Six APV25 chips require six dedicated DAQ channels.

- *Common I^2C settings* While five of the six APV25 chips involved require the same settings for their I^2C interface, one chip requires different settings.

All objects shown are created in the free store at runtime. They are destroyed at program shutdown or when the configuration is reloaded.

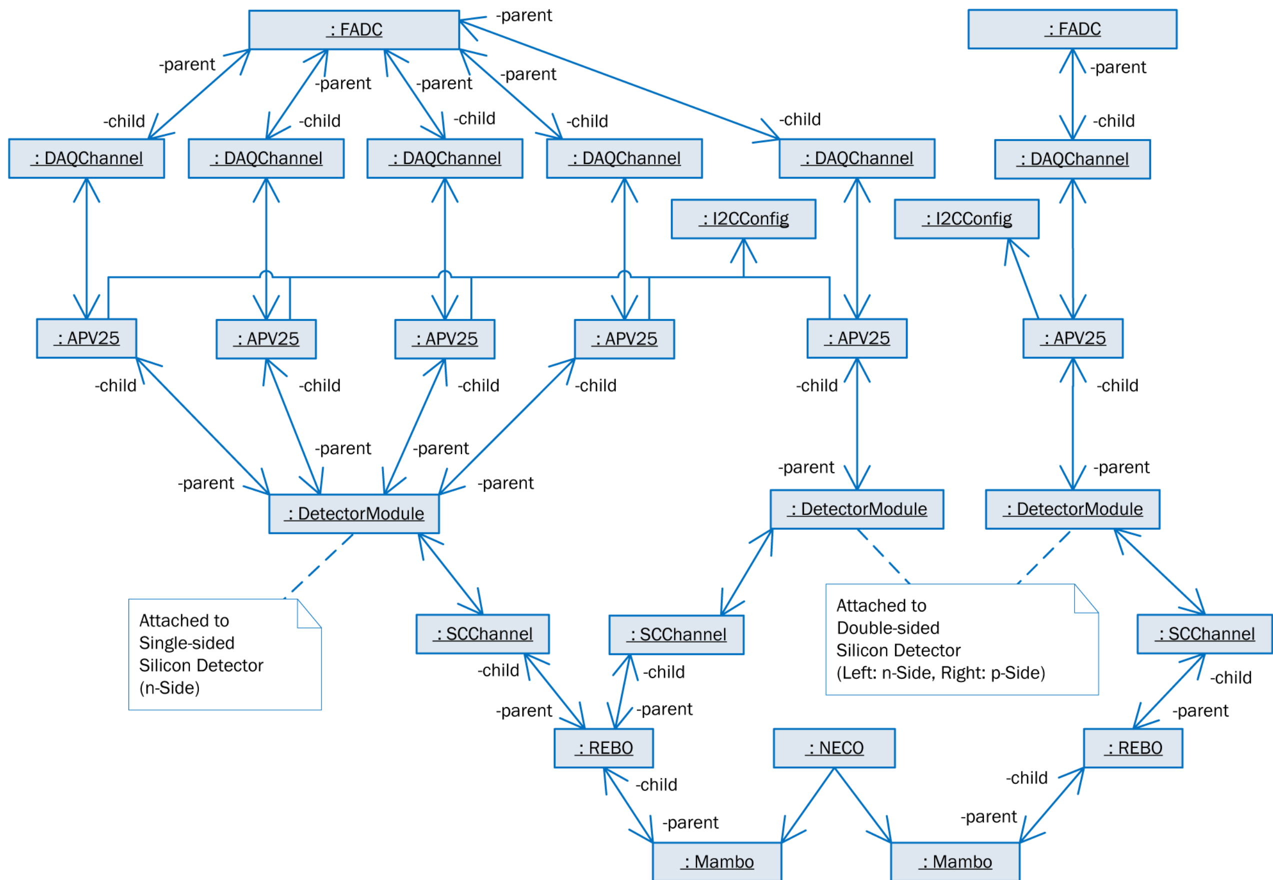


Figure 5.5: *Objects and Object Relationships for a Typical Configuration (UML Object Diagram)*

5.6 Model-View-Controller

The core architecture of TuxDAQ is based on the *Model-View-Controller (MVC)* [33] triad of classes. MVC uncouples application logic and presentation. It consists of three types of objects:

- The *Model* encapsulates the core data and functionality (i.e. the *business logic*) of the application. The model is not aware of any concrete view or controller objects. In other words, it is not aware of how it is presented to the user and manipulated by them.

- The *View* is responsible for the presentation of the model. It knows the model and makes it accessible for the user. The model allows to register all views available so that it can notify them every time the model state changes. Once notified, a view retrieves the new state and all data it requires from the model

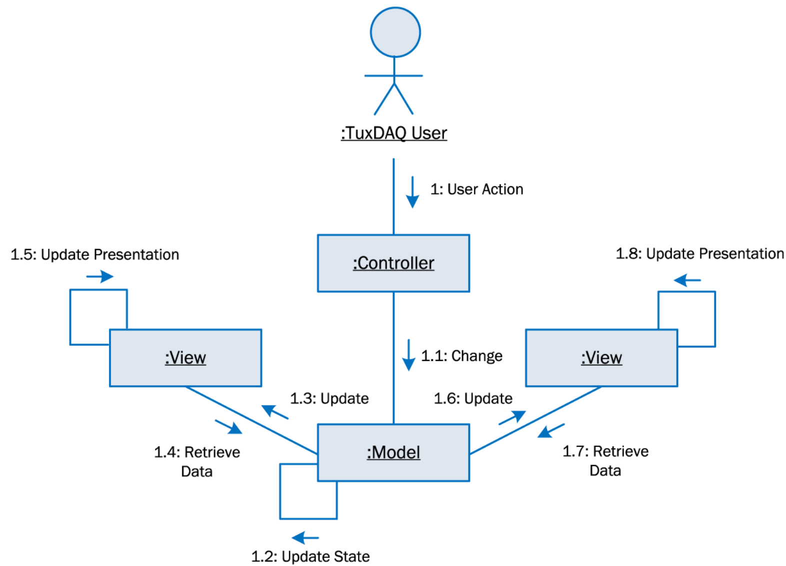


Figure 5.6: *Interaction between MVC Objects (UML Communication Diagram)*

to perform an update of the presentation. As per Requirements 3.4.8/1 and 3.4.8/3, TuxDAQ comprises two separate views, one representing the GUI and one for console outputs.

- The *Controller* creates the model object and all views at application startup. As a next step, it registers the views at the model. During operation, it processes user actions and links them to appropriate updates of the model.

Fig. 5.6 gives an overview of the interaction between objects in MVC, as described above. The objects are uncoupled: Consequently, changes to the model can affect any number of views without requiring the changed model to know details of the views. When the model changes its state, all registered views are notified and updated automatically. This is an example of the *Observer* [32] design pattern.

The design described above has the following advantages:

- The TuxDAQ model can be represented by various types of presentation with a minimum of developmental effort.

- All views are synchronised automatically with the model. Consequently, the user can work with different views at the same time.

- New views can be added without the need to modify existing code except for the registration of the new view at the model.

Fig. 5.7 gives an overview of all classes involved. The classes *Model*, *View* and *Controller* are reusable abstract classes which define a robust interface for the basic interaction between the MVC objects. *TuxDModel* and *TuxDController* are TuxDAQ-specific and represent the model and the controller. *TuxDGUIView* and *TuxDGUIQt* implement the GUI, and *TuxDConsoleView* is intended for the console interface. *TuxDView* defines the data members and member functions that are common to all TuxDAQ-specific views.

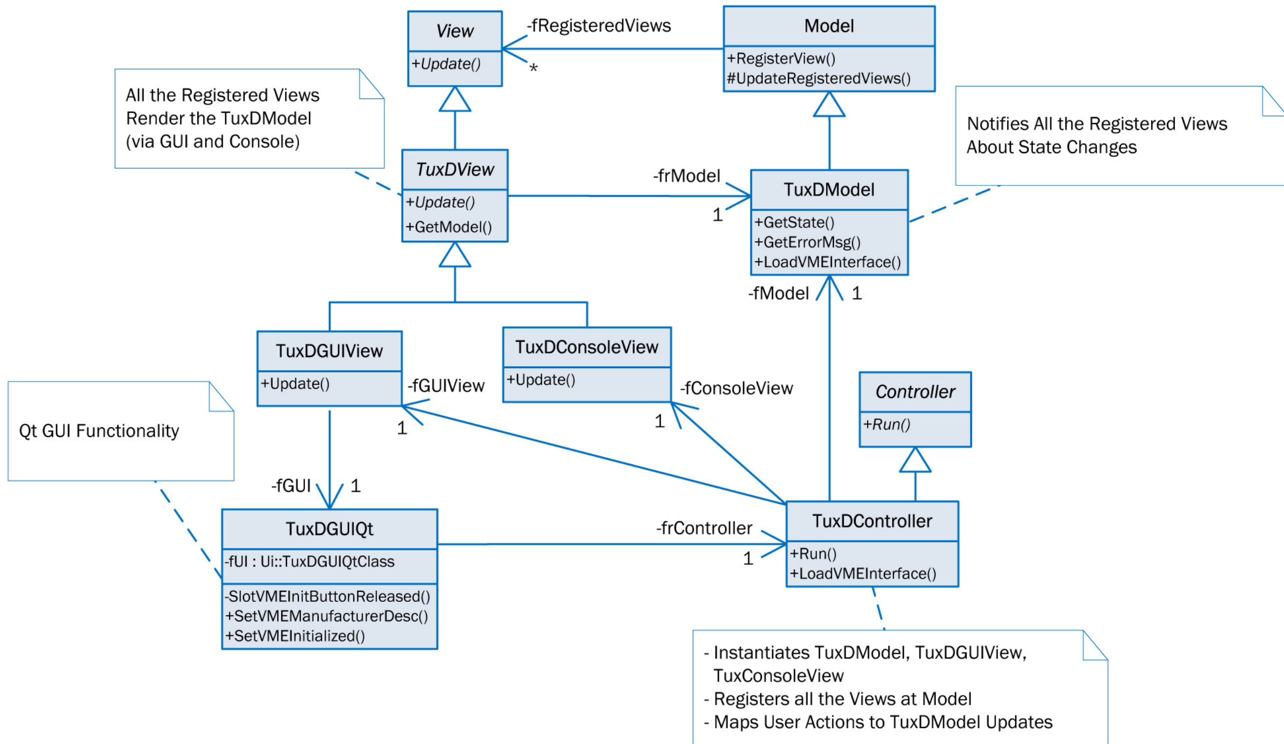


Figure 5.7: *Model-View-Controller Class Hierarchy (UML Class Diagram)*

As shown in Listing 5.2, an instance of TuxDController is created at program startup. After that, the controller instantiates the model (TuxDModel) as well as one view for each user interface (TuxDGUIView and TuxConsoleView) and registers them at the model. TuxDController provides a member function Run() to start the 'main loop' of the application.

The constructor of TuxDGUIView creates an instance of TuxDGUIQt, which contains Qt-specific user interface data and behaviour. It recognises user actions in the GUI by means of the Qt-specific *signals and slots* mechanism and calls appropriate member functions of the TuxDController object. The relationship between TuxDGUIView and TuxDGUIQt is described in more detail in Section 5.6.2.

In TuxDAQ, two views are registered at the model, specifically one TuxDGUIView object and one TuxD-ConsoleView object. A view can be registered at the model using the member function Model::RegisterView(). Every time the model state changes, Model::UpdateRegisteredViews() is invoked (see Listing 5.4[73]). This function is not aware of implementation-specific details of the registered views, but invokes their concrete member functions TuxDGUIView::Update() and TuxDConsoleView::Update() via the View interface. Each registered view retrieves the state and all data required from the model to perform an update of the presentation.

The class Model depends on an abstraction only, namely the View interface. Therefore, the concrete type of view can be changed without any need to change the model. The relationship between Model and View presented here is an example for the *Strategy* [32] design pattern.

[73]The code listing shows how to use an *iterator* to access the elements of an aggregate object that contains View pointers only. Invoking `this->GetRegisteredViews()` returns a reference to this aggregate object. This technique is established in C++ and corresponds to the *Iterator* [32] design pattern.

Listing 5.4: *Notifying All Views Registered at the Model*

```
1 void Model::UpdateRegisteredViews()
2 {
3    std::vector<View*>::iterator viewsIterator;
4    for (viewsIterator = (this->GetRegisteredViews()).begin();
5          viewsIterator != (this->GetRegisteredViews()).end(); viewsIterator++) {
6       (*viewsIterator)->Update();
7    }
8 }
```

In the following sections, the model and view components are described in more detail.

5.6.1 The TuxDAQ Model

The TuxDAQ model has wide-ranging responsibilities:

1. Encapsulation of the functional core of TuxDAQ
2. Application configuration
3. Data Acquisition (DAQ) management

To improve maintainability and reusability and to keep all modules compact, these responsibilities were assigned to *two* separate classes, namely *TuxDModel* and *DAQControl*. As shown in Fig. 5.8, TuxDModel covers Responsibility 1, and DAQControl covers the other responsibilities[74]. These classes are associated by means of a *delegation* relationship: The TuxDModel object (*delegator*) dispatches every message that is out of its scope to the DAQControl object (*delegate*) [48]. For example, TuxDModel provides a member function TuxDModel::StartRun(), which is intended for starting a DAQ run (see Section 2.2.1). As this task is in the scope of DAQControl (Responsibility 3), TuxDModel dispatches the request to DAQControl (i.e., DAQControl also provides a member function StartRun(), which is invoked by TuxDModel::StartRun()). The clients of TuxDModel are not aware of this delegation.

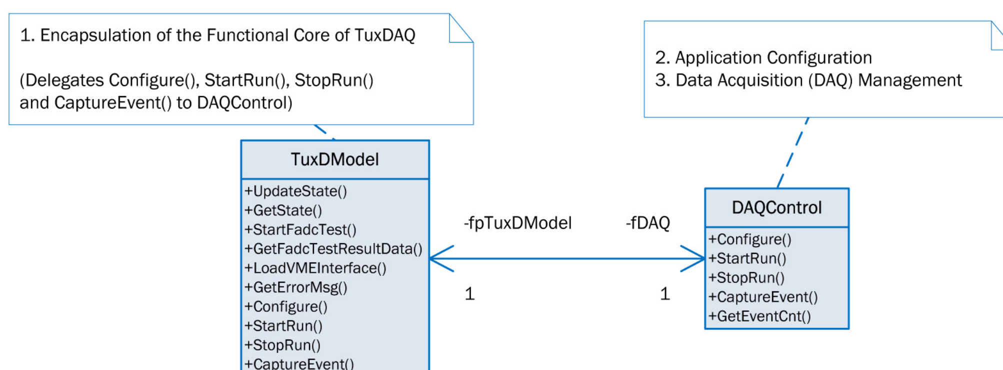


Figure 5.8: *TuxDModel and DAQControl (UML Class Diagram)*

[74]DAQControl has two responsibilities and therefore violates the Single Responsibility Principle (SRP). This compromise was made to keep the class library compact.

With TuxDModel's responsibility being different from the responsibilities of DAQControl, both classes were associated by means of delegation instead of inheritance[75].

According to its responsibility, TuxDModel provides various member functions to

- update and retrieve the model state,
- start an FADC+PROC test and retrieve its results (see Requirement 3.4.5/8),
- initialise the VME interface,
- retrieve an error message in case of an error.

Additionally, TuxDModel provides all the functions dispatched to DAQControl, e.g. StartRun().

TuxDModel was designed as a *Finite State Machine (FSM)* to realise the functional core of TuxDAQ: The class consists of a finite number of states and transitions between those states, which are executed under certain conditions (referred to as *guard conditions*).

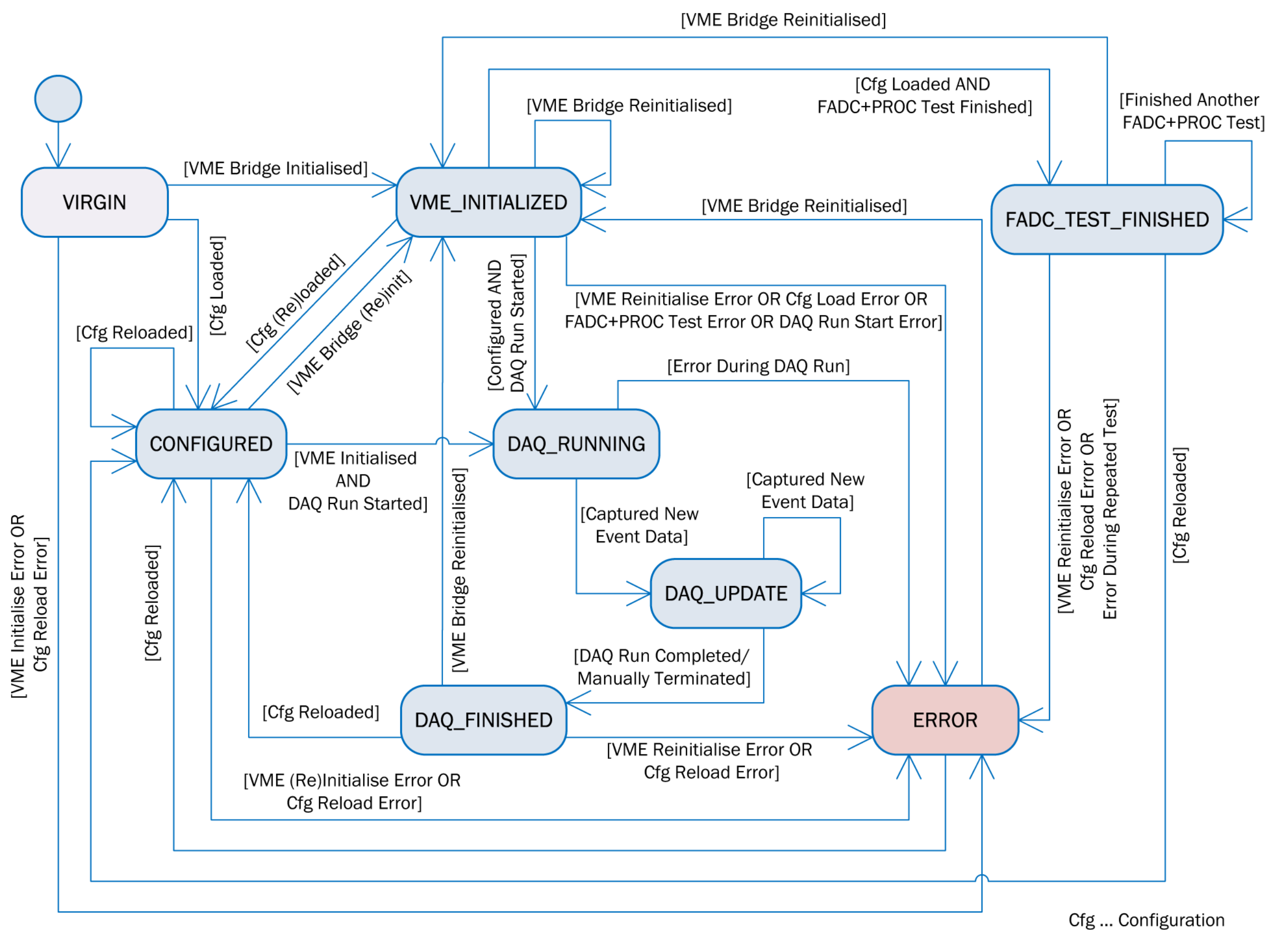


Figure 5.9: *Class TuxDModel as Finite State Machine (UML State Diagram)*

[75]"Inheritance should only be used when the generalisation relationship is semantically valid. Inheritance means that each instance of a subclass truly is an instance of the superclass; thus all operations and attributes of the superclass must uniformly apply to the subclass." [49]

As shown in Fig. 5.9, TuxDModel consists of the following states:

- *The initial state* This state represents the point in time when the TuxDModel constructor is called. The constructor, however, goes into the state VIRGIN.
- *VIRGIN* In this state, the model has been created but user activity has yet to take place.
- *VME_INITIALIZED* The model goes into this state after the VME bridge has been initialised successfully. This can be triggered manually by the user (the GUI provides a button to load the VME interface), or automatically when the user triggers the start of a DAQ run.
- *CONFIGURED* This state is reached after the model has loaded a specific configuration based on a configuration file. The user can select and load such a file manually by means of a dedicated GUI element. This feature allows for exploration of a specific configuration (see Requirement 3.4.5/1). However, the configuration file is loaded automatically in any case at the start of a DAQ run. Once the model goes into this state, the free store contains an object constellation as shown in Fig. 5.5. The model allows the user to decide, whether the VME bridge is initialised before a configuration file is loaded or vice versa (if done manually).
- *DAQ_RUNNING* When the user triggers the start of a DAQ run, the model initialises the VME bridge (if not initialised manually before), loads the configuration file desired and attempts to initialise the entire readout system based on this file. If successful, the model goes into the state DAQ_RUNNING and starts capturing events periodically, as described in Section 2.2.3.
- *DAQ_UPDATE* This state indicates the capture of a new event from the readout system. Before the model goes into this state, it saves all event data to an output file, with the user being able to decide if data is to be written to such a file ahead of the DAQ run. When the views registered at the model recognise this state, they update their elements indicating the progress of the run, e.g. the event counter (see Requirements 3.4.5/9 and 3.4.5/10). Additionally, the GUI contains a graphical plot widget presenting the captured APV25 data. It retrieves this data from the model as soon as the model goes into the state DAQ_UPDATE.
- *DAQ_FINISHED* The model goes into this state after a DAQ run has concluded. Ahead of this, the readout system is stopped and all output files are closed.
- *FADC_TEST_FINISHED* This state is reached after an FADC+PROC test has been concluded successfully. All registered views retrieve the test data from the model and present it to the user.
- *ERROR* TuxDModel catches all possible exceptions[76] that might occur in TuxDAQ during runtime. Once an exception is caught, TuxDModel reads out the respective error message and goes into the state ERROR. If the clients of TuxDModel, i.e. the views, recognise this state, they can read out the error message with a dedicated member function and present it to the user.

[76] Apart from exceptions of type std::bad_alloc, which are caught in main().

- *The final state* This state represents the point in time when the TuxDModel object is destroyed. This is the case at the application shutdown, which can be initiated by the user at any time regardless of the model state. For the sake of clarity, the diagram does not show the final state and according transitions.

The class DAQControl provides member functions to

- parse a configuration file,
- load the configuration into the free store,
- manage a DAQ run,
- retrieve data about the progress of a DAQ run.

The configuration file parser works with configuration files in the legacy format of the APVDAQ readout application (see Requirement 3.4.4/1).

5.6.2 The TuxDAQ Views

The user interfaces of TuxDAQ were designed and implemented as follows:

- *The console interface* As a console interface is only desirable and not mandatory for TuxDAQ (see Requirement 3.4.8/3), its implementation was not considered high priority. Therefore, the according class TuxDConsoleView was prepared only for future development. At this point in time, it therefore merely presents the results of an FADC+PROC test on the console.
- *The GUI* As mentioned in Section 5.3, the GUI of TuxDAQ is based on Qt. To achieve greater reusability, separate classes were introduced to represent the GUI: TuxDGUIView provides an Update() function and acts as an observer of the model (like TuxDConsoleView). It is associated with another class TuxDGUIQt, which contains all Qt-specific code (see Fig. 5.7). With this separation, TuxDGUIView can be reused in case Qt is replaced by another GUI technology in the future. In short, TuxDGUIQt is the only class of TuxDAQ that depends on Qt.

With *Qt Designer*, Qt provides a graphical tool for designing and building GUIs. With this tool, it is possible to compose a GUI and assign variable names to respective widgets easily. It is also possible to save a GUI form in a so-called *UI file*. This UI file is referenced within a separate *Qt project file*. The Qt-specific build tool *qmake* uses both files as input files and produces a C++ header file containing the definition of an appropriate class representing the GUI (see Fig. 5.10)[77].

As shown in Fig. 5.7, TuxDGUIView is associated with *Ui::TuxDGUIQtClass*, (see attribute of TuxDGUIView). This class is automatically generated and updated by qmake based on the procedure mentioned above. Within the code of class TuxDGUIView, it is possible to access all variables representing the GUI via a reference to the Ui::TuxDGUIQtClass instance. With this separation, the GUI can be updated with Qt designer (or replaced by another GUI) without any impact on the user code (which is contained in TuxDGUIQt).

[77] In addition to the scenario explained, the Qt project file contains all files relevant for building the whole application (e.g. header files, source code files, etc.). Based on the Qt project file, qmake not only produces the file shown, but also an appropriate *makefile* containing all information required to build the application with GNU Make [50]. See Appendix B for an overview of the directories and files in the TuxDAQ repository.

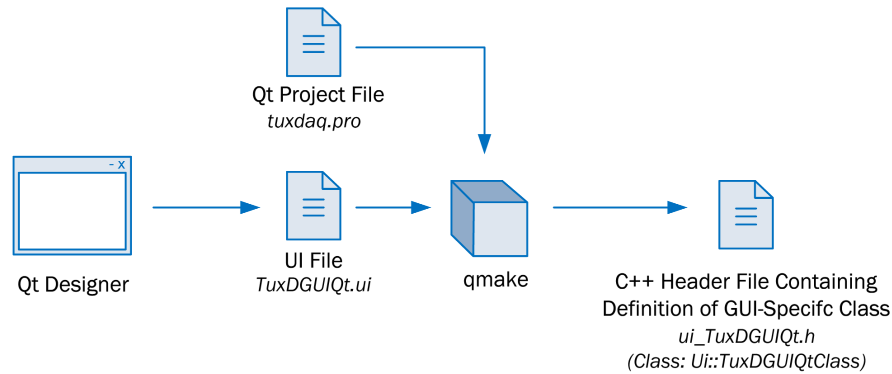


Figure 5.10: *Designing and Building the TuxDAQ GUI with Qt*

In Qt, user actions are processed with a mechanism called *signals and slots*: Every widget can send a signal triggered by a specific event, e.g. when a button is released after it has been pressed. This signal can be associated with an appropriate slot by the application programmer. A slot is a function which implements the desired behaviour in response to the triggering signal. All slots are implemented in TuxDGUIQt. For example, this class provides a slot *SlotVMEInitButtonReleased()* (see Fig. 5.7). It is associated with a signal that can be sent by a button dedicated to the initialisation of the VME bridge[78].

5.6.3 A Typical Scenario

To summarise the interaction between the TuxDAQ-specific MVC components, Fig. 5.11 presents a sequence diagram for the following scenario:

- The user presses a GUI button in order to initialise the VME bridge manually, for example to test the connection to the VME controller. This button is assigned to the slot SlotVMEInitButtonReleased() of class TuxDGUIQt, which notifies the controller about the user action.
- The controller dispatches this request to the model, which instructs an appropriate factory to create a new VMEInterface object by invoking the Create() member function of the factory. The constructor of the new VMEInterface object initialises the VME bridge, and the model retrieves a pointer to this newly created object.
- As a next step, the model updates its state to VME_INITIALIZED and notifies all registered views, namely one TuxDConsoleView instance and one TuxDGUIView instance.
- The TuxDConsoleView instance retrieves the state but does not react accordingly - as mentioned above, currently it provides FADC+PROC test functionality only.
- First of all, the TuxDGUIView instance retrieves the state and subsequently all data required to update the graphical presentation from the model. Finally, it updates the Qt GUI, as it modifies the text of a dedicated GUI label and an icon indicating that the VME bridge has been initialised.

Note that the VME initialisation succeeds and no other errors occur in the given scenario.

[78]The assignment of signals to slots is done in the constructor of TuxDGUIQt.

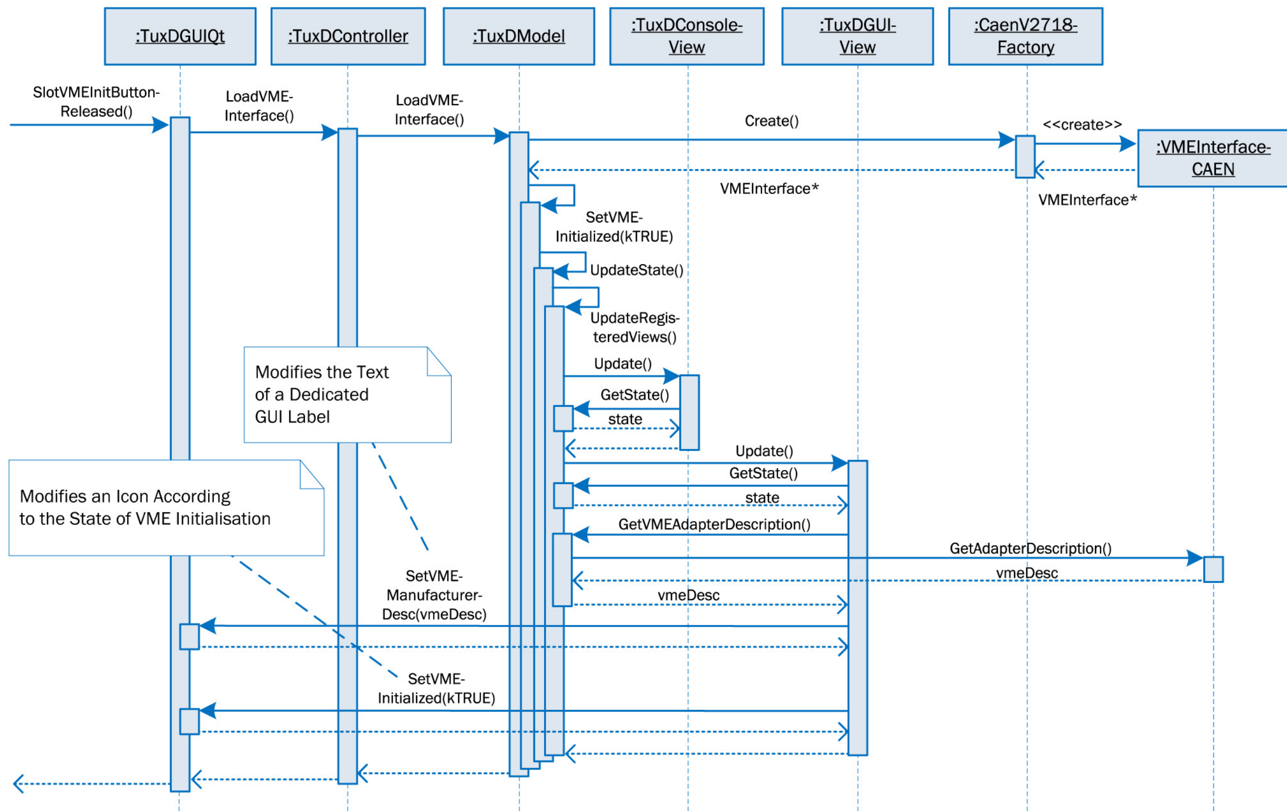


Figure 5.11: *Typical Scenario for MVC Interaction: The User Presses a Button in Order to Initialise the VME Bridge (UML Sequence Diagram)*

5.7 Interaction between TuxDAQ and TuxOA

As described in Section 3.2, the online processing and analysis of acquired data will be performed by a separate program, namely TuxOA. As per the requirements of Section 3.4.6, the running of this program is optional, in case a user wants to analyse the data captured.

The MVC architecture of TuxDAQ allows for a simple way to implement the communication between TuxDAQ and TuxOA: As TuxOA - albeit in a separate program - simply represents another view observing the TuxDAQ model, the following solution would be appropriate:

- TuxDAQ establishes a connection to TuxOA if desired by the user. For example, a *TCP/IP socket* connection can be used. This solution would even allow the running of both programs on different hosts in a common TCP/IP network. Usually, TCP/IP sockets are inherently supported by Linux-based operating systems.
- The TuxDAQ model needs to notify TuxOA if its state changes. To best way to achieve this, would be to implement a subclass of TuxDView, e.g. *TuxOAView*, which provides an Update() function to send the model state to TuxOA via the socket connection. This solution has no impact on the code of any other classes (except for DAQControl, see below).
- TuxOA acts like a view of TuxDAQ: It is aware of all model states and makes them accessible to the user,

retrieving the state and all data required from the TuxDAQ model via the socket connection. On TuxDAQ side, this communication is also implemented in TuxOA::Update(). To access the data acquired from the readout system (required by TuxOA), TuxOAView needs to be associated with DAQControl. The latter has to be extended by a public member function to retrieve the captured event data[79].

5.8 Logging[80]

As per Requirements 3.4.5/(9, 11) and 3.4.7/(8, 9, 10), *log statements* had to be inserted into the program code of TuxDAQ. Instead of including a self-developed logging API, the *Apache log4cxx framework* was used (see Section 5.3). This framework allows to:

- *"Control which log statements are output at arbitrary granularity"* [40]. The application developer can define a *level of granularity* for each log statement, and the user is able to define up to which level of granularity log messages are output per configuration file. By this means, the logging output can be adapted to the specific needs of every user and the amount of log data reduced.

- *Manage logging with configuration files* Based on configuration files, log statements contained in the code of the application can be enabled or disabled selectively. Moreover, the format of logging output can be defined, and the output can be sent to multiple targets, e.g. to the standard console and to a *log file*.

- *Incorporate logging at a low performance cost* Log4cxx was developed with the claim to be "fast and flexible" [40].

5.8.1 Loggers, Appenders and Layouts

To enable log4cxx-based logging, the relevant program has to retrieve at least one logger from the log4cxx API. A logger is a named entity responsible for processing *logging requests*. A logging request is a statement by which a programmer requests logging output.

It is possible to set up a logger hierarchy, which provides a specific inheritance mechanism. This hierarchy is based on the logger name as follows:

> "A logger is said to be an *ancestor* of another logger if its name followed by a dot is a prefix of the *descendant* logger name. A logger is said to be a *parent* of a *child* logger if there are no ancestors between itself and the descendant logger." [40]

For example, the logger named 'hephy' is a parent of the logger named 'hephy.el' and an ancestor of 'hephy.el.tuxdaq'.

Every logger may be assigned a *level*. Considering that they are ordered, the pre-defined levels are:

[79] For TuxDAQ as a self-contained program, it has not been necessary to access this data outside of DAQControl.
[80] The information provided here about Apache log4cxx is based on [40].

$$TRACE < DEBUG < INFO < WARN < ERROR < FATAL$$

If a given logger is not assigned a level, it inherits one from the closest ancestor with an assigned level[81].

Logging requests can also be made at a certain level, which is depicted in Listing 5.5: in line 1, a logger named ‘hephy.el’ is retrieved from the log4cxx API. In line 2, the macro LOG4CXX_INFO is used to invoke an appropriate member function of the logger instance. In this way, a logging request is made at level INFO. Similarly to LOG4CXX_INFO , the log4cxx API provides macros for all levels, e.g. LOG4CXX_WARN.

Listing 5.5: *Log4cxx: Retrieving a Logger and Making a Logging Request at Level INFO*

```
1  log4cxx::LoggerPtr logger(log4cxx::Logger::getLogger("hephy.el"));
2  LOG4CXX_INFO(logger, "This is a log message.")
```

The macros described support “short-circuiting” [40] if the logging request is disabled as per the following selection rule:

> “A logging request is said to be enabled, if its level is higher than or equal to the level of its logger. Otherwise, the request is said to be disabled.” [40]

The level of a logging request is defined through the selection of the appropriate macro by the programmer; the level of a logger can either be defined with a dedicated API function or with a configuration file setting. In the latter scenario, a user can define a level below which all requests relating to a specific logger are disabled. Therefore, the granularity of logging information can be controlled without impacting or recompiling the program code.

Log4cxx allows logging requests to be output to multiple destinations. These are referred to as *appenders*. The framework provides appenders for the console, files, GUI components, remote socket servers, NT Event loggers, and remote UNIX Syslog daemons. A logger can be associated with one or more appenders based on the following rule[82]:

> “Each enabled logging request for a given logger will be forwarded to all the appenders in that logger as well as the appenders higher in the hierarchy.” [40]

For example, if a console appender is added to a logger named ‘hephy’, and a file appender is added to a logger named ‘hephy.el’, all enabled logging requests of ‘hephy.el’ will be output to the console and to a file. The enabled logging requests of ‘hephy’, however, will only print on the console. The behaviour described is also referred to as *appender additivity*.

The output format of log messages can be defined by associating a *layout* with an appender. For TuxDAQ, the PatternLayout was used for all appenders. This layout allows to specify the output format similar to the printf() function of the C language.

[81] At the top of the hierarchy, there is the so-called *root logger*, which always exists and has an assigned level.
[82] This default behaviour can be changed with the so-called *additivity flag*.

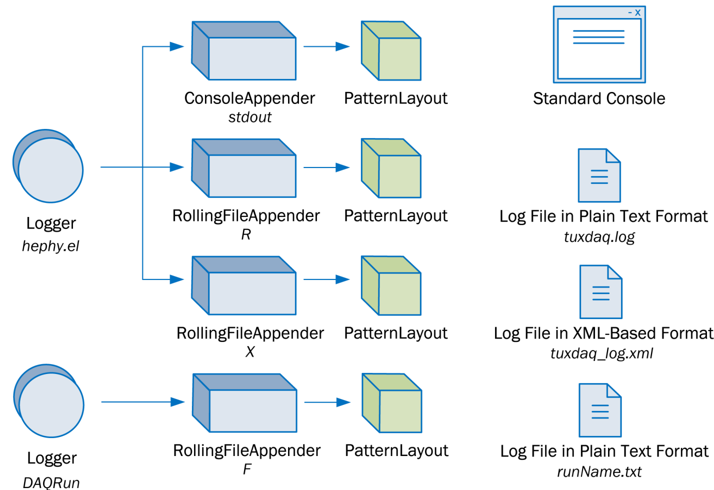


Figure 5.12: *The Logging Strategy for TuxDAQ, Defined in a Configuration File*

5.8.2 Logging Configuration

In TuxDAQ, every class retrieves a logger with a name that corresponds to the fully qualified name of the class from the log4cxx API. For example, the class hephy::el::NecoVME[83] is assigned a logger named 'hephy.el.NecoVME'. This strategy makes it easy to identify the origin of a log message.

Together with appenders and layouts, the loggers are configured in a configuration file. Fig. 5.12 shows how this configuration file affects the logging concept for TuxDAQ:

- *Application-wide and DAQ run-specific logging* In the configuration file, two different loggers are defined: *hephy.el* and *DAQRun*. The former is intended for application-wide logging, i.e. it logs the order of all events of interest from application startup to shutdown. Due to the inheritance of logger levels and appender additivity, the settings for the logger hephy.el apply to all the class-specific loggers. All TuxDAQ-specific classes belong to the namespaces hephy::el or hephy::el::tuxdaq, which relate to logger prefixes hephy.el or hephy.el.tuxdaq. The DAQRun logger is intended to be used for one log file produced by the class DAQControl for each DAQ run.

- *Application-wide logging: different output targets* Application-wide logging is forwarded to three output targets: the standard console (to fulfil Requirement 3.4.7/9), a log file in plain text format and one in an XML-based format (both to fulfil Requirement 3.4.7/10).

- *DAQ run-specific logging: special format* The format of DAQ run-specific log files is compatible with the HEPHY offline analysis tool (see Requirement 3.4.2/3).

- *Logger levels* In addition to the definitions described above, the levels of both loggers are defined. For standard usage, the level of logger hephy.el should be set to INFO. It can be changed to DEBUG or even TRACE if the program is to be scrutinised more extensively by a developer.

Appendix B provides a detailed overview of the TuxDAQ repository, which also describes the names of files and directories related to logging.

[83] See Section 5.3.1 for a description of the namespaces introduced in TuxDAQ.

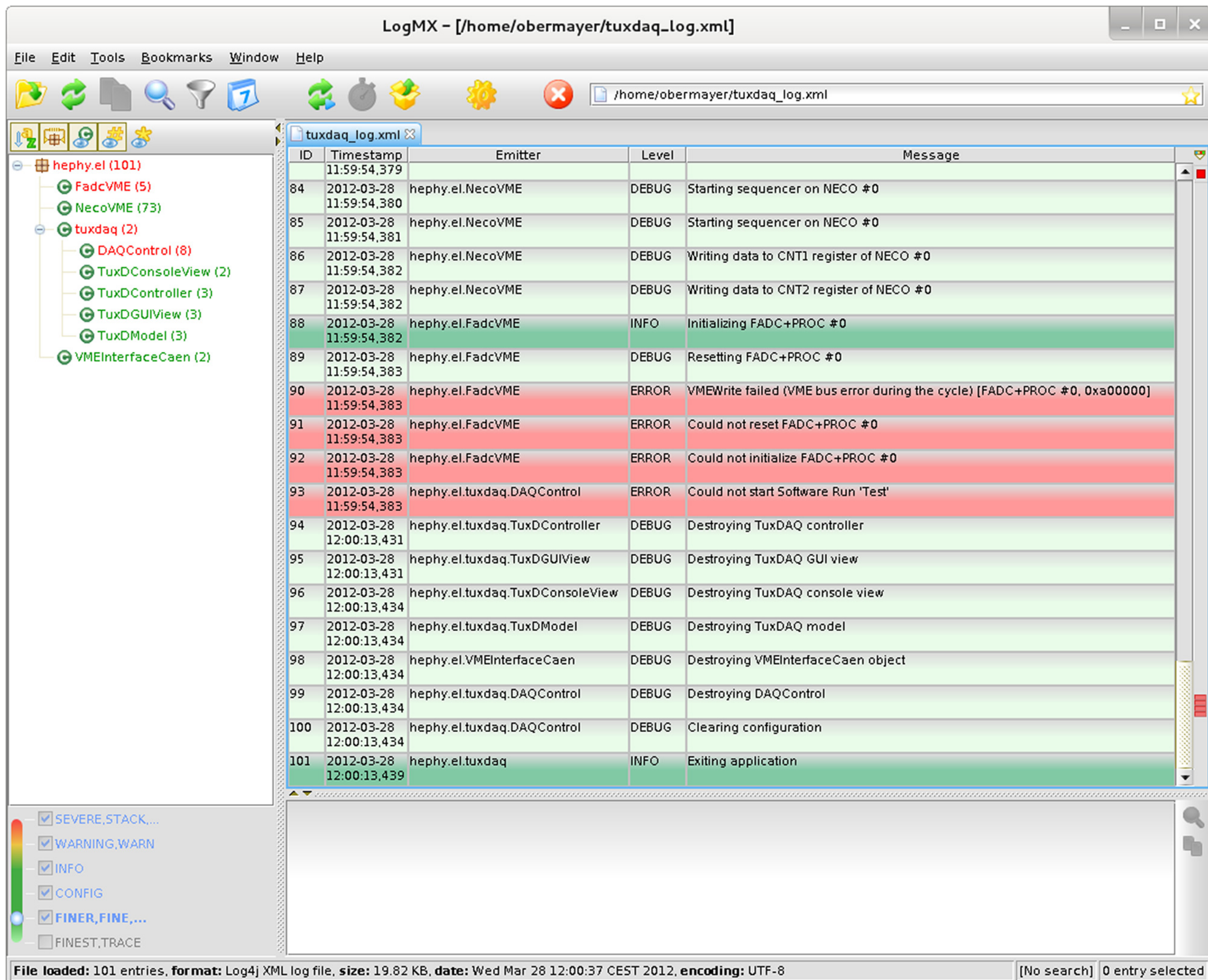


Figure 5.13: *Log File Analysis of an Error Scenario with LogMX: A run named 'Test' was started. The start of the run did not succeed as the initialisation of the FADC+PROC #0 VME board (attempting to reset the board) failed. The according stack trace messages are shown in red colour.*

5.8.3 Log File Analysis with a Log File Viewer

As described above, application-wide logging is not only written to a file in plain text format, but also to a file in XML-based format[84]. This file contains the same log messages as the plain text file, but can be loaded into a log file viewer, which allows for comfortable, detailed log analysis by the user.

Fig 5.13 provides a screenshot of a log file analysis with LogMX [51], a commercial log file viewer for multiple log file formats and operating systems. It can be seen that log messages are coloured according to their level, which simplifies log file analysis. It is also possible to filter out messages belonging to a specific logger which maps to a specific class in TuxDAQ. In the lower left corner of the window, the user can control a threshold level below which log messages are filtered out[85].

[84]Log4cxx uses the popular format of log4j.

[85]As LogMX allows to process multiple log file formats, the level names are different to those of log4cxx by default. While this could be changed, it was not done in this instance as the program was only evaluated.

6

Application Overview for Users

This chapter provides a short introduction to TuxDAQ from a user perspective. It is assumed that the application has been installed properly and that all system requirements are fulfilled. These are described in Section 7.3.

6.1 Starting TuxDAQ

TuxDAQ can be started in a Linux console with the following command:

```
../tuxdaq_directory > ./tuxdaq
```

If required, the application can be started with a parameter defining the type of VME bridge (see Section 2.1.7) to be used:

- `./tuxdaq caen` instructs TuxDAQ to use the CAEN A2818/V2718 VME bridge
- `./tuxdaq sis` instructs TuxDAQ to use the SiS 1100e/3104 VME bridge

If no parameter is given, TuxDAQ uses the CAEN VME bridge.

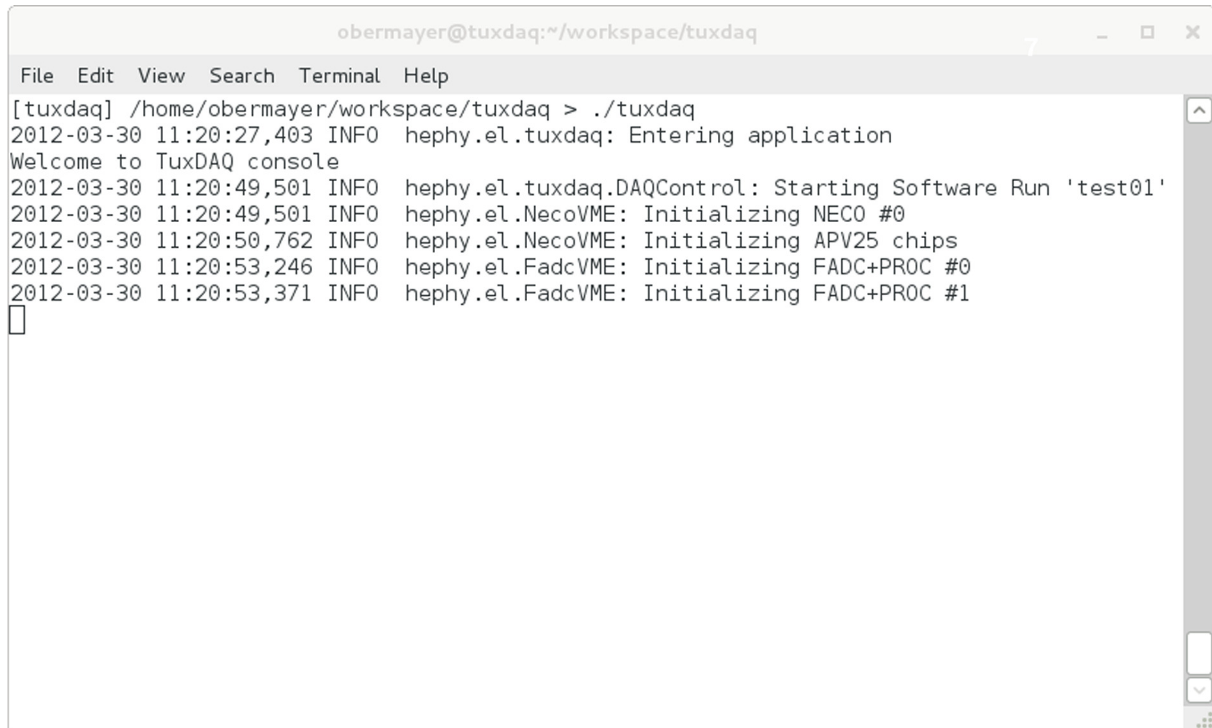


Figure 6.1: *Screenshot: The Linux Console During TuxDAQ Operation*

When started, TuxDAQ loads its GUI. The console is also used; and - as described in Section 5.8.2 - the order of all significant events is logged to the console. This feature can be disabled by modifying the logging configuration file accordingly[86]. Fig. 6.1 shows a screenshot of the console after TuxDAQ had been started and a run named ‘test01’ was triggered.

6.2 Different GUI Tabs

The TuxDAQ GUI comprises three tabs:

1. *The DAQ tab (see Fig.* 6.2*)* This tab provides several widgets to set up, start, monitor and stop a DAQ run.

2. *The front-end electronics tab (see Fig.* 6.3*)* Once a configuration file has been loaded, the structure of the readout system’s front-end components defined within that file can be explored. Additionally, a plot widget allows the monitoring of the APV25-related data captured during a DAQ run.

3. *The back-end electronics tab (see Fig.* 6.4*)* Similar to the front-end electronics tab, this tab allows the exploration of the back-end components of the readout system defined within a configuration file. In addition to this, the VME bridge can be initialised manually and an FADC+PROC test can be performed via this tab.

[86]See Appendix B for an overview of the directories and files in the TuxDAQ repository.

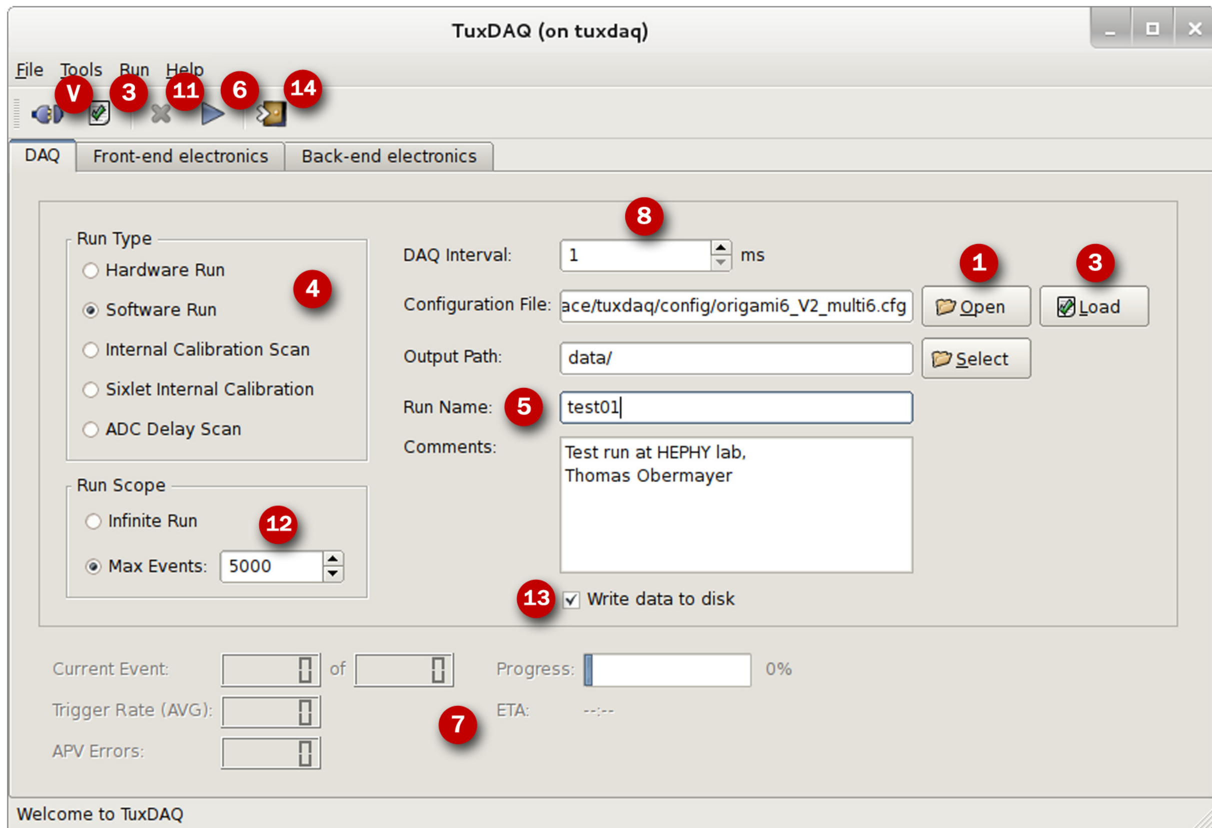


Figure 6.2: *TuxDAQ Screenshot: The DAQ Tab*

6.3 Performing a DAQ Run

Below, the typical scenario for performing a DAQ run[87] is described. The numbers and letters between parentheses refer to the respective labels in Figures 6.2, 6.3 and 6.4.

After starting up the application, the GUI is loaded. Now the user can select a configuration file (1), which defines the structure of the readout hardware (front-end and back-end) as well as certain configuration parameters. Having selected an appropriate file, the arrangement of both front-end and back-end electronics defined within may be verified with tree widgets (2) in the front-end and back-end electronics tabs. If the given configuration settings have to be changed, the configuration file can be edited during runtime with an external text editor. The modified file can then be reloaded (3) without a restart of the application.

In the next step, the user can select the type of DAQ run that is to be performed (4). Depending on whether or not acquired data should be written to a file, an output path, a run name and respective comments can be entered (5).

Once an appropriate configuration file has been loaded, a run type selected and data output set up properly, a DAQ run can be initiated with the 'Start Run' button (6): the application performs an initialisation of the VME bridge and all involved hardware components based on configuration file settings. For this purpose, it addresses NECO and FADC+PROC boards via the VME bridge. Slow controls, intended for the front-end, are

[87]Only the hardware and software run have been implemented so far. If any of the other run types is selected, the user is notified about this fact.

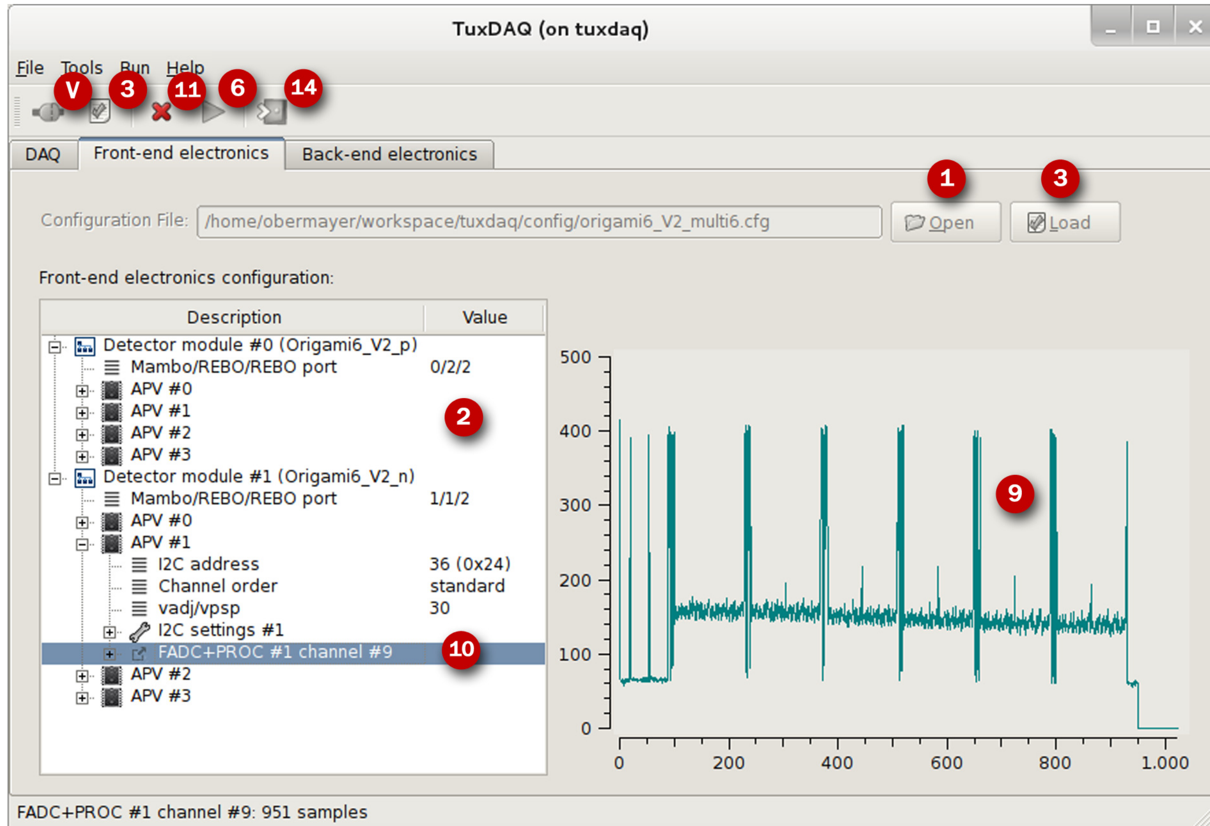


Figure 6.3: *TuxDAQ Screenshot: The Front-end Electronics Tab*

handed over to the NECO controller, which distributes them to Dock boxes accordingly. They are forwarded to respective detector modules. This procedure is logged to the console (see Fig. 6.1) and to two different log files (see Section 5.8.2).

Once the setup of hardware components has finished, the acquisition of data from the FADC+PROC boards starts. In addition to the console window, several GUI widgets (7) monitor the ongoing DAQ run. Further to the actual event rate (rate of allowed APV25 triggers), the user is presented with the number of the currently captured event, the number of occurred APV errors and the ETA (estimated completion time).

A timer-based architecture allows the user to interact with the GUI during a DAQ run. Therefore, a *DAQ interval*[88] can be defined (8), which corresponds to a period of time during which data is acquired cyclically. A plot widget (9) in the front-end tab shows data recently captured from an FADC+PROC channel (DAQ channel)[89], which can be selected by the user (10).

Generally, a DAQ run concludes after the ‘Stop Run’ button (11) has been pressed. In addition to that, the user may define a maximum number of events to be captured (12). In this case, the DAQ run terminates automatically as soon as the desired set of events has been recorded. This setting can also be changed during an ongoing DAQ process. If the value is zero or the option ’Infinite Run’ has been selected, the DAQ run does

[88]Typically (and at minimum) 1 ms

[89]In Fig. 6.3, raw FADC data of a captured particle hit is visualised by the plot widget (abscissa: time in clock periods; ordinate: amplitude in ADC counts). The plot shows the data regarding one respective event, where six consecutive samples have been captured following one APV25 trigger signal (chip operated in multi-peak mode). As mentioned in Section 2.1.2, for each of the six samples a digital header is followed by 128 values of analogue strip data.

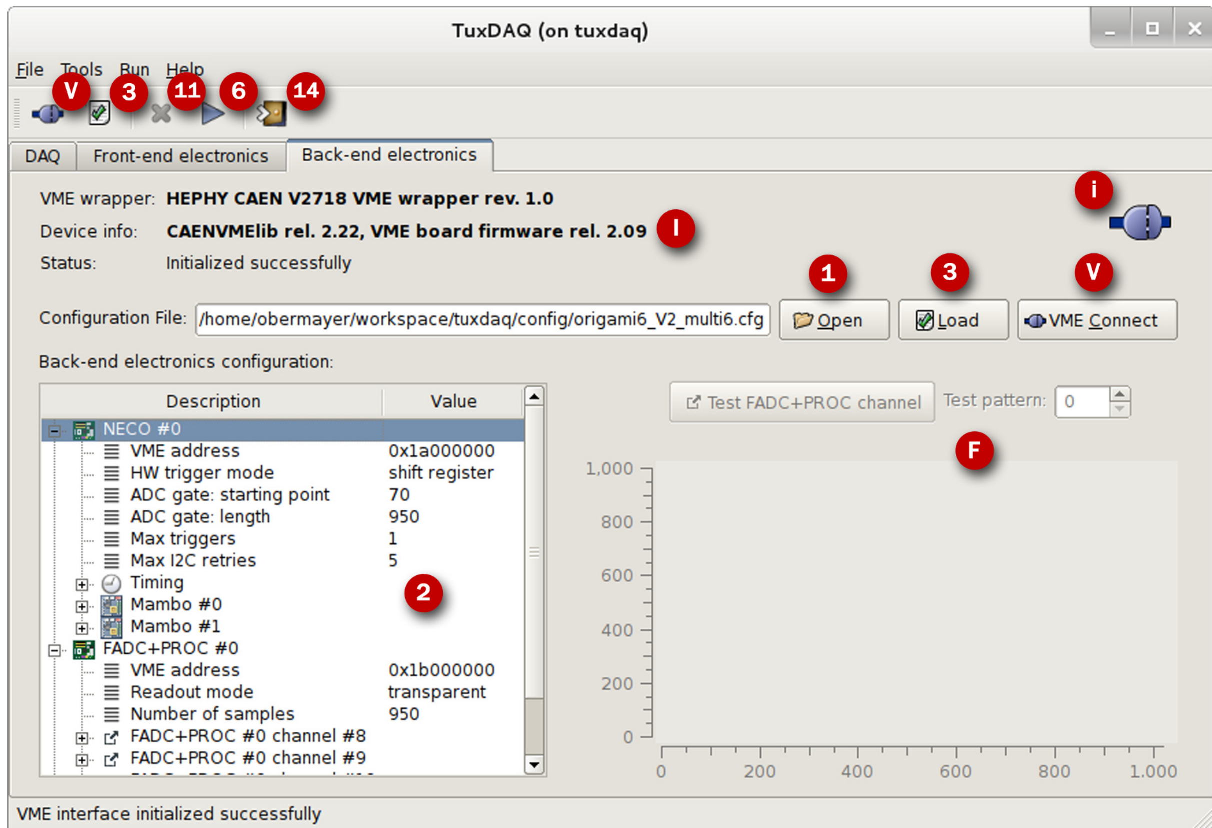


Figure 6.4: *TuxDAQ Screenshot: The Back-end Electronics Tab*

not conclude automatically. Instead, the user has to terminate the process manually.

All data acquired is written to disk, if desired (13). TuxDAQ allows the execution of several successive DAQ runs without the need for a program restart. The user can quit the application by pressing the 'Quit' button (14).

6.4 VME Bridge Initialisation

It is possible to initialise the VME bridge manually (V) prior to the start of a DAQ run. This allows for testing the proper functioning of the VME bridge without having to perform a DAQ run. A dedicated icon (i) indicates whether the VME bridge has been initialised (manually or automatically at the start of a DAQ run). The main information about the VME bridge is presented at the top of the back-end electronics tab (I).

6.5 FADC+PROC Test

Similar to the APVDAQ readout application, TuxDAQ provides an FADC+PROC test function (F), which is described in Section 2.2.4.

6.6 Error Treatment

Any errors are reported to the user by means of an error dialogue in the GUI. Additionally, the error message and the according stack trace are logged to the console and log files (see Fig. 5.13).

6.7 The Configuration File

TuxDAQ configuration files have the format of APVDAQ configuration files, which are described in Section 2.2.5.

6.8 Data Output Files

TuxDAQ produces the following data output files:

- *Run-specific output files* To be compatible with the existing offline analysis tool at HEPHY, TuxDAQ produces the same set of data output files as the APVDAQ readout application (see Section 2.2.6). This refers to file types, names, sizes and formats.
- *Log files* As described in Section 5.8.2, all the log messages (output on the console) are also written to log files in plain text and an XML-based format.

7

Software Environment

This chapter describes the software environment of TuxDAQ. At first, experiences with different hardware platforms and operating systems during application development are described. Following this, an overview of all the software tools involved is presented. Finally, the prerequisites for the successful operation of TuxDAQ are listed.

7.1 Operating Systems and VME Libraries

Prior to the development of TuxDAQ, a new, state-of-the-art PC comprising an ASUS P8H67-V motherboard [52] with a 64-bit Central Processing Unit (CPU) was acquired. According to the original plan, Debian 6 (64-bit) [53] with Linux kernel 2.6.32 should have been used as an operating system.

Unfortunately, tests of the CAEN A2818/V2718 VME bridge (see Section 2.1.7) with the *A2818 Linux driver 1.16* and the *CAENVMELib library 2.30.2* led to inexplicable errors when performing VME block transfers; they did not succeed and the system crashed, which required a hardware reset of the VME controller. Through extensive research the conclusion was reached that the PCI implementation of the motherboard was not supported by the operating system.

Consequently, the modern PC mentioned above was replaced by older hardware to ensure compatibility with Linux, and Scientific Linux CERN 5 (32-bit) [54] was installed[90]. With this setup, tests of the CAEN VME bridge were successful. As the next step, the VME bridge was tested on Debian 6 (64-bit) and Scientific Linux CERN 5 (64-bit) with positive results.

[90]Experience reports confirmed the proper functioning of the CAEN VME bridge with this Linux distribution.

Finally, it was decided to use CERN Scientific Linux 5 (64-bit) for further development and operation. This operating system is widely used in the related research area and long-term supported by CERN. It is based on Red Hat Enterprise Linux 5 [55]. In addition to the CAEN VME bridge, the SiS 1100e/3104 VME bridge (see Section 2.1.7) with *Linux driver 2.13* was tested successfully on this operating system.

7.2 Development Tools

The most important software tools and libraries used for TuxDAQ *development* were[91]:

- *The GNU toolchain* This established collection of free programming tools includes, among other tools, the GNU Compiler Collection (GCC) with C++ support 4.1.2 [56], the GNU Project Debugger (GDB) 7.0.1 [57] and GNU Make 3.81 [50].
- *Apache Subversion (SVN) 1.6.11* This is an open source version control system [58]. The TuxDAQ SVN repository is described in more detail in Appendix B.
- *Doxygen 1.4.7* As described in Section 5.1, this documentation system allows for the automated generating of documentation based on source code comments [37].
- *Eclipse IDE for C/C++ Developers (Indigo release)* This is an integrated software development platform which also provides several plugins supporting the tools mentioned above [59].
- *Additional C++ development libraries* As described in Section 5.3, the following C++ libraries were installed and used: Qt 4.2.1 [45], Qwt 6.0 [46], Apache log4cxx 0.10.0 [40] and SimpleIni 4.14 [43].

7.3 Prerequisites for TuxDAQ Operation

TuxDAQ requires the following software environment for successful *operation*:

- Scientific Linux CERN 5 (64-bit, with X11) or a compatible operating system. TuxDAQ should also compile and run on 32-bit operating systems. This was tested successfully in an early project phase; no more test runs, however, were conducted at later stages.
- Qt 4.2.1 (or higher) and Qwt 6.0 (or higher)
- Apache log4cxx 0.10.0 (or higher)
- The drivers and libraries for all VME bridges TuxDAQ will use.

[91] Except for Apache log4cxx, SimpleIni, Qwt and the Eclipse IDE, all software tools and libraries are part of the Scientific Linux CERN 5 distribution.

8 Conclusion

The APVDAQ readout application proved to be successful as a DAQ software application for the prototype of the Belle II SVD readout system. However, due to shortcomings in its design and documentation, maintenance and further development of the program turned out to be problematic. Moreover, in the research field of high energy physics, a Linux-based solution has been demanded, due to the Windows-based APVDAQ readout application being impractical.

In this thesis, a software development process was presented, which allowed for time efficient development of a new DAQ software application providing a sophisticated, object-oriented, long-term maintainable architecture: based on an all-embracing set of software requirements, the prototype readout system was analysed in regards to the identification of class/object candidates, their attributes and behaviour in a dedicated Object-Oriented Analysis (OOA) phase. Following this, in the Object-Oriented Design (OOD) phase the conceptual OOA model was revised with the objective of a software architecture that will not rot over time. Obeying basic design principles and following popular design patterns were vital strategies for achieving this goal. Furthermore, the Object-Oriented Programming (OOP) methodology aimed at a C++ program that was easy to read, test and debug, and could be adapted to changing requirements.

For the new software solution, the tasks of the APVDAQ readout application were assigned to two separate Linux-based programs: TuxDAQ and TuxOA. Thereby, the online processing and analysis of acquired data (covered by TuxOA) was separated from the DAQ functionality (covered by TuxDAQ) to improve runtime-flexibility and performance.

This thesis presented an overview of TuxDAQ's advanced design and implementation. Particular attention was paid to two types of software reuse: on one hand, existing standard components were reused in order to reduce development effort, cost and risk. On the other hand, the architecture of TuxDAQ was designed in a way that allowed for the extraction of self-contained components, to be reused for the Belle II Experiment and other projects in the area of high energy physics research.

An appropriate subset of the Unified Modeling Language (UML) was used to describe the software structure and behaviour in brief. In addition to this, an exhaustive web-based documentation of the TuxDAQ API was created - vital for a framework of reusable components.

For TuxOA, the system boundaries were defined and an approach for the communication between TuxDAQ and TuxOA was proposed.

Most notably, the focus on proper software engineering and documentation in this project brought out a stable, flexible, state-of-the-art system which can be distinguished by its high level of reusability and maintainability. With these properties, TuxDAQ has a very good chance to prove itself at HEPHY in the long run, and parts of the TuxDAQ framework allow for a reduction of development effort for the Belle II Experiment.

However, some limitations are worth noting. Due to time constraints, not all software requirements could be fulfilled. Future work should therefore include the implementation of a *run for internal calibration* (see Requirement 3.4.5/6) and an *ADC delay scan* (see Requirement 3.4.5/7). In addition to this, TuxOA has to be implemented, which also requires some enhancements of TuxDAQ (see Requirements in Section 3.4.6). If desired, the console interface of TuxDAQ (see Requirement 3.4.8/3), which has already been prepared, could be advanced.

A

Traceability

A.1 Source Traceability

In addition to the rationales associated with requirements (see Section 3.4), these requirements can be linked to the stakeholders who proposed them. By these means, the origin of requirements is traceable and involved people can be consulted if a requirements change is proposed [20].

In Table A.1, the proposed priority and origin of each requirement are presented. The two-letter acronyms in the column 'Origin' refer to involved engineers at the HEPHY Department of Electronics:

- *MF* Markus Friedl
- *CI* Christian Irmler
- *TO* Thomas Obermayer

Most of the requirements evolved from the project definition[92], which implied that TuxDAQ was to be developed on basis of the APVDAQ readout application (see Section 2.2).

[92]Preliminary meeting (MF, CI, TO) on 26/11/2010, project kick-off in 04/2011

Table A.1: TuxDAQ Source Traceability

Req. ID	Short Text	Origin
3.4.2/1	Compatibility with the Belle-II SVD prototype readout system	Project definition (MF, CI)
3.4.2/2	Multiple VME bridges	Project definition (MF, CI)
3.4.2/3	Interoperability with HEPHY offline analysis tool	Requirements meeting (MF, CI, TO), 12/12/2011 [18]
3.4.2/4	Linux compliance	Project definition (MF, CI)
3.4.3/1	Main purpose of TuxDAQ	Project definition (MF, CI)
3.4.3/2	Usage at external research facilities	[3]
3.4.3/3	Reusability for the Belle-II Experiment	Project definition (MF, CI)
3.4.4/1	Configurability by means of a configuration file	Project definition (MF, CI)
3.4.4/2	No sensors and zones necessary	Requirements meeting (CI, TO), 9/11/2011 [18]
3.4.4/3	Selection of VME bridge	Requirements meeting (MF, CI, TO), 12/12/2011 [18]
3.4.5/1	Exploration of loaded configuration	Requirements meeting (CI, TO), 9/11/2011 [18]
3.4.5/2	No online data processing and analysis	Requirements meeting (MF, CI, TO), 23/10/2011 [18]
3.4.5/3	Hardware run	Project definition (MF, CI)
3.4.5/4	Software run	Project definition (MF, CI)
3.4.5/5	Typical workflow	Project definition (MF, CI)
3.4.5/6	Run for internal calibration	Requirements meeting (CI, TO), 6/7/2011
3.4.5/7	ADC Delay Scan	Requirements meeting (CI, TO), 6/7/2011
3.4.5/8	FADC+PROC test	Requirements meeting (CI, TO), 14/7/2011
3.4.5/9	Progress monitoring of active run	Project definition (MF, CI)
3.4.5/10	Estimation of run conclusion	Project definition (MF, CI)
3.4.5/11	Offline run reproduction	Project definition (MF, CI)
3.4.6/1	Stand-alone operability	Project definition (CI)
3.4.6/2	Automatic TuxOA invocation	TO
3.4.6/3	Collaboration with TuxOA	Project definition (CI)
3.4.7/1	Configuration file validation	Several requirements meetings (CI, TO)
3.4.7/2	User input validation	Several requirements meetings (CI, TO)
3.4.7/3	Low-level communication link error handling	Project definition (MF, CI)
3.4.7/4	DAQ error handling	Project definition (MF, CI)
3.4.7/5	Data storage error handling	Project definition (MF, CI)

Continued on next page

Table A.1: TuxDAQ Source Traceability

Req. ID	Short Text	Origin
3.4.7/6	Recovery from failures	Requirements meeting (MF, CI, TO), 12/12/2011 [18]
3.4.7/7	Error notifications	Project definition (MF, CI)
3.4.7/8	Stack trace on error	MF
3.4.7/9	Error information on the screen	Project definition (MF, CI)
3.4.7/10	Offline error tracking	Project definition (MF, CI)
3.4.8/1	Graphical User Interface (GUI)	Project definition (CI)
3.4.8/2	GUI tasks	Several requirements meetings (CI, TO)
3.4.8/3	Console interface	Several requirements meetings (CI, TO)
3.4.8/4	GUI as default user interface	Requirements meeting (MF, CI, TO), 12/12/2012 [18]
3.4.9/1	TuxDAQ as open source project	Project definition (MF, CI)
3.4.9/2	Usage of free software components	Project definition (MF, CI)
3.4.9/3	Object-oriented design and implementation	Project definition (CI)
3.4.9/4	C++ as programming language	Project definition (MF, CI)
3.4.9/5	Compatibility with the CERN ROOT framework	Project definition (MF, CI)
3.4.9/6	Minimum performance	Project definition (MF, CI)
3.4.9/7	Maximum file size	Requirements meeting (MF, CI, TO), 12/12/2012 [18]
3.4.9/8	System evolution	Project definition (MF, CI)
3.4.10/1	Application Programming Interface (API) reference manual	Project definition (CI)
3.4.10/2	Inline source code comments	Project definition (CI)

A.2 Requirements Traceability

Table A.2 describes which requirements are affected by one particular requirement. This information is useful to assess the impact of proposed requirements changes on the whole software engineering process [20].

Table A.2: TuxDAQ Requirements Traceability

Req. ID	Short Text	Affects Requirement(s)
3.4.2/1	Compatibility with the Belle-II SVD prototype readout system	3.4.2/2, 3.4.3/(1, 2), 3.4.4/1, 3.4.5/(2, 3, 4, 6, 7, 8), 3.4.7/(3, 4), 3.4.9/(6, 8)

Continued on next page

Table A.2: TuxDAQ Requirements Traceability

Req. ID	Short Text	Affects Requirement(s)
3.4.2/2	Multiple VME bridges	3.4.4/3, 3.4.9/4
3.4.2/3	Interoperability with HEPHY offline analysis tool	3.4.9/7
3.4.2/4	Linux compliance	3.4.9/4
3.4.3/1	Main purpose of TuxDAQ	3.4.8/2
3.4.3/2	Usage at external research facilities	
3.4.3/3	Reusability for the Belle-II Experiment	3.4.9/3, 3.4.10/1
3.4.4/1	Configurability by means of a configuration file	3.4.5/1, 3.4.7/1, 3.4.8/2
3.4.4/2	No sensors and zones necessary	
3.4.4/3	Selection of VME bridge	
3.4.5/1	Exploration of loaded configuration	3.4.8/2
3.4.5/2	No online data processing and analysis	3.4.4/2, 3.4.5/(3, 4)
3.4.5/3	Hardware run	3.4.5/(5, 9, 10, 11), 3.4.7/2, 3.4.8/2
3.4.5/4	Software run	3.4.5/(5, 9, 10, 11), 3.4.7/2, 3.4.8/2
3.4.5/5	Typical workflow	3.4.8/2
3.4.5/6	Run for internal calibration	3.4.5/(9, 10, 11), 3.4.7/2, 3.4.8/2
3.4.5/7	ADC Delay Scan	3.4.5/(9, 10, 11), 3.4.7/2, 3.4.8/2
3.4.5/8	FADC+PROC test	3.4.7/2, 3.4.8/2
3.4.5/9	Progress monitoring of active run	3.4.8/2
3.4.5/10	Estimation of run conclusion	3.4.8/2
3.4.5/11	Offline run reproduction	
3.4.6/1	Stand-alone operability	
3.4.6/2	Automatic TuxOA invocation	
3.4.6/3	Collaboration with TuxOA	3.4.8/2
3.4.7/1	Configuration file validation	3.4.7/(6, 7, 8, 9, 10)
3.4.7/2	User input validation	3.4.7/(6, 7, 8, 9, 10)
3.4.7/3	Low-level communication link error handling	3.4.7/(6, 7, 8, 9, 10)
3.4.7/4	DAQ error handling	3.4.7/(6, 7, 8, 9, 10)
3.4.7/5	Data storage error handling	3.4.7/(6, 7, 8, 9, 10)
3.4.7/6	Recovery from failures	
3.4.7/7	Error notifications	
3.4.7/8	Stack trace on error	3.4.7/(9, 10), 3.4.8/3
3.4.7/9	Error information on the screen	3.4.8/2
3.4.7/10	Offline error tracking	
3.4.8/1	Graphical User Interface (GUI)	3.4.8/(2, 4)

Continued on next page

Table A.2: TuxDAQ Requirements Traceability

Req. ID	Short Text	Affects Requirement(s)
3.4.8/2	GUI tasks	3.4.8/3
3.4.8/3	Console interface	
3.4.8/4	GUI as default user interface	
3.4.9/1	TuxDAQ as open source project	
3.4.9/2	Usage of free software components	3.4.9/(4, 5)
3.4.9/3	Object-oriented design and implementation	3.4.9/4
3.4.9/4	C++ as programming language	3.4.9/5
3.4.9/5	Compatibility with the CERN ROOT framework	
3.4.9/6	Minimum performance	
3.4.9/7	Maximum file size	
3.4.9/8	System evolution	
3.4.10/1	Application Programming Interface (API) reference manual	
3.4.10/2	Inline source code comments	

A.3 Design Traceability

Table A.3 describes which classes/files are affected by one particular requirement. This information is useful to assess the impact of proposed requirements changes on the software design and implementation [20].

Table A.3: TuxDAQ Design Traceability

Req. ID	Short Text	Affects Class(es)/Files
3.4.2/1	Compatibility with the Belle-II SVD prototype readout system	APV25, CFGException, DAQChannel, DAQControl, DetectorModule, EventHeader, FADC, FADCException, FadcVME, FadcVMEException, I2CConfig, Mambo, NECO, NECOException, NecoVME, NecoVMEException, REBO, SCChannel, TuxDConsoleView, TuxDController, TuxDGUIQt, TuxDGUIView, TuxDModel
3.4.2/2	Multiple VME bridges	CaenException, CaenV2718Factory, FadcVME, NecoVME, SIS3104Factory, SISException, VMEException, VMEInterface, VMEInterfaceCaen, VMEInterfaceFactory, VMEInterfaceSIS

Continued on next page

Table A.3: TuxDAQ Design Traceability

Req. ID	Short Text	Affects Class(es)/Files
3.4.2/3	Interoperability with HEPHY offline analysis tool	DAQControl, EventHeader
3.4.2/4	Linux compliance	hephytypes.h, TimeTools
3.4.3/1	Main purpose of TuxDAQ	
3.4.3/2	Usage at external research facilities	
3.4.3/3	Reusability for the Belle-II Experiment	APV25, CaenException, CaenV2718Factory, CFGException, Component, Composite, Controller, DAQChannel, DAQControl, DetectorModule, EventHeader, FADC, FADCException, FadcVME, FadcVMEException, hephytypes.h, I2CConfig, Mambo, Model, NECO, NecoVME, NecoVMEException, REBO, SCChannel, SIS3104Factory, SISException, StringTools, TimeTools, View, VMEException, VMEInterface, VMEInterfaceCaen, VMEInterfaceFactory, VMEInterfaceSIS
3.4.4/1	Configurability by means of a configuration file	CFGException.h, DAQControl, TuxDConsoleView, TuxDController, TuxDGUIQt, TuxDGUIView, TuxDModel
3.4.4/2	No sensors and zones necessary	CFGException.h, DAQControl
3.4.4/3	Selection of VME bridge	main.cpp
3.4.5/1	Exploration of loaded configuration	DAQControl, TuxDConsoleView, TuxDGUIQt, TuxDGUIView, TuxDModel
3.4.5/2	No online data processing and analysis	DAQControl, TuxDConsoleView, TuxDGUIQt, TuxDGUIView, TuxDModel
3.4.5/3	Hardware run	APV25, CFGException, DAQChannel, DAQControl, DetectorModule, FADC, FADCException, FadcVME, FadcVMEException, I2CConfig, Mambo, NECO, NECOException, NecoVME, NecoVMEException, REBO, SCChannel, TuxDConsoleView, TuxDController, TuxDGUIQt, TuxDGUIView, TuxDModel, VMEException, VMEInterface
3.4.5/4	Software run	see Req. 3.4.5/3
3.4.5/5	Typical workflow	test
3.4.5/6	Run for internal calibration	not implemented yet
3.4.5/7	ADC Delay Scan	not implemented yet
3.4.5/8	FADC+PROC test	see Req. 3.4.5/3

Continued on next page

Table A.3: TuxDAQ Design Traceability

Req. ID	Short Text	Affects Class(es)/Files
3.4.5/9	Progress monitoring of active run	DAQControl, FadcVME, FadcVMEException, NecoVME, NecoVMEException, TuxDConsole-View, TuxDController, TuxDGUIQt, TuxDGUIView, TuxDModel, VMEException, VMEInterface
3.4.5/10	Estimation of run conclusion	DAQControl, TuxDController, TuxDConsoleView, TuxDGUIQt, TuxDGUIView, TuxDModel
3.4.5/11	Offline run reproduction	DAQControl, FadcVME, NecoVME
3.4.6/1	Stand-alone operability	not implemented yet
3.4.6/2	Automatic TuxOA invocation	not implemented yet
3.4.6/3	Collaboration with TuxOA	not implemented yet
3.4.7/1	Configuration file validation	DAQControl, TuxDGUIQt, TuxDGUIView, TuxD-Model
3.4.7/2	User input validation	see Req. 3.4.7/1
3.4.7/3	Low-level communication link error handling	DAQControl, FadcVME, NecoVME, TuxDGUIQt, TuxDGUIView, TuxDModel, VMEException, VMEInterface, VMEInterfaceCaen, VMEInterface-SIS
3.4.7/4	DAQ error handling	DAQControl, FadcVME, NecoVME, TuxDController, TuxDGUIQt, TuxDGUIView, TuxDModel, VMEEx-ception, VMEInterface, VMEInterfaceCaen, VMEIn-terfaceSIS
3.4.7/5	Data storage error handling	DAQControl, TuxDController, TuxDGUIQt, TuxDGUIView, TuxDModel
3.4.7/6	Recovery from failures	see Req. 3.4.7/(1, 2, 3, 4, 5)
3.4.7/7	Error notifications	see Req. 3.4.7/(1, 2, 3, 4, 5)
3.4.7/8	Stack trace on error	see Req. 3.4.7/(1, 2, 3, 4, 5)
3.4.7/9	Error information on the screen	see Req. 3.4.7/(1, 2, 3, 4, 5), config/logger.config
3.4.7/10	Offline error tracking	see Req. 3.4.7/9
3.4.8/1	Graphical User Interface (GUI)	TuxDController, TuxDGUIQt, TuxDGUIView
3.4.8/2	GUI tasks	TuxDGUIQt, TuxDGUIView
3.4.8/3	Console interface	TuxDConsoleView, TuxDController
3.4.8/4	GUI as default user interface	Req. 3.4.8/4 b) not implemented yet, for the rest: TuxDController
3.4.9/1	TuxDAQ as open source project	all classes/files
3.4.9/2	Usage of free software components	all classes/files

Continued on next page

Table A.3: TuxDAQ Design Traceability

Req. ID	Short Text	Affects Class(es)/Files
3.4.9/3	Object-oriented design and implementation	all classes/files
3.4.9/4	C++ as programming language	all classes/files
3.4.9/5	Compatibility with the CERN ROOT framework	all classes/files
3.4.9/6	Minimum performance	all classes/files
3.4.9/7	Maximum file size	DAQControl, EventHeader
3.4.9/8	System evolution	APV25, CFGException, DAQChannel, DAQControl, DetectorModule, FADC, FADCException, FadcVME, FadcVMEException, hephytypes.h, Mambo, NECO, NECOException, NecoVME, NecoVMEException, REBO, SCChannel, TuxDGUIQt
3.4.10/1	Application Programming Interface (API) reference manual	
3.4.10/2	Inline source code comments	all classes/files

B

The TuxDAQ Repository

The software described in this thesis and the associated documentation (see Section 5.1) are available through the TuxDAQ SVN repository [60] (rev. 153). As shown in Fig. B.1, the most important folders and files are:

- *config/* This folder is reserved for configuration files. In addition to various TuxDAQ configuration files (e.g. origami_v2_multi6.cfg), the following files are available:
  - *fadc_test_data.csv, fadc_test_pedestals.csv* In these files, the FADC+PROC test patterns (see Section 6.5) are defined.
  - *logger.config* This file is intended for the configuration of TuxDAQ logging, as described in Section 5.8.2.
- *data/* This is the default output folder for the files TuxDAQ produces in a DAQ run (see Section 6.8). DAQ run-specific log files are also stored here.
- *doc/* This folder contains all files related to the project documentation of TuxDAQ.
  - *html/* This folder provides an exhaustive web-based documentation of the TuxDAQ API, which is described in Section 5.1.
    - *index.html* This file contains the front page of the TuxDAQ API documentation.
  - *tuxdaq_documentation.Doxyfile* This is the configuration file of the Doxygen documentation system [37], which is described in Section 5.1.
- *log/* This is the output folder for application-wide log files, which are described in Section 5.8.2.
- *Makefile* This file is related to the GNU Make [50] build utility. It specifies how to derive the target program from the source code of TuxDAQ. See Footnote 77 for further details.

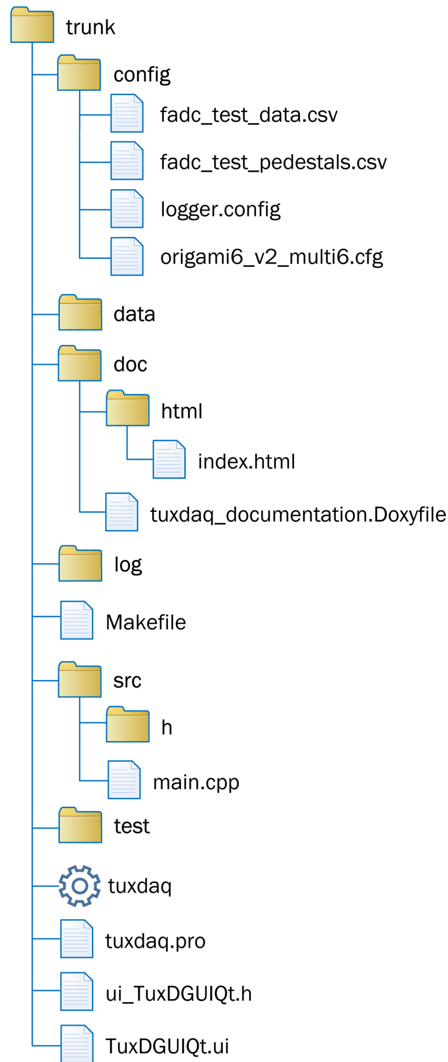


Figure B.1: *The TuxDAQ SVN Repository (Revision 153)*

- *src/* This folder contains the C++ source code files of TuxDAQ.
  - *h/* This folder contains the header files of TuxDAQ.
  - *main.cpp* This file provides the TuxDAQ's main() function, which indicates the starting point of the program.
- *test/* In this folder, test drivers for the most critical classes of the TuxDAQ framework are provided.
- *tuxdaq* This is the TuxDAQ executable binary file.
- *tuxdaq.pro* This is the Qt-specific project file of TuxDAQ. See Footnote 77 for further details.
- *ui_TuxDGUIQt.h, TuxDGUIQt.ui* These files are described in Section 5.6.2.

Acronyms

ADC	Analogue-To-Digital Converter
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
CASE	Computer-Aided Software Engineering
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
CRC	Class Responsibility Collaboration
DAQ	Data Acquisition
DIP	Dependency Inversion Principle
DSSD	Double-Sided Silicon Strip Detector
ETA	Estimated Time of Arrival
FADC	Flash Analogue-to-Digital Converter
FADC+PROC	Flash Analogue-to-Digital Converter and Data Processing
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GCC	GNU Compiler Collection
GDB	GNU Project Debugger
GoF	Gang of Four
GUI	Graphical User Interface
HEPHY	Institute of High Energy Physics of the Austrian Academy of Sciences
IDE	Integrated Development Environment
ISP	Interface Segregation Principle
LGPL	GNU Lesser General Public License
LSP	Liskov Substitution Principle
MIP	Minimum Ionising Particle
MIT	Massachusetts Institute of Technology
MVC	Model-View-Controller
NECO	New Controller
NI	National Instruments

OA	Online Analysis
OCP	Open Closed Principle
OOA	Object-Oriented Analysis
OOD	Object-Oriented Design
OOP	Object-Oriented Programming
PC	Personal Computer
RAII	Resource Acquisition Is Initialisation
RAM	Random Access Memory
REBO	Repeater Board
SC	Slow Control
SiS	Struck Innovative Systeme
SRP	Single Responsibility Principle
SRS	System Requirements Specification
SSD	Silicon Strip Detector
SSSD	Single-Sided Silicon Strip Detector
SVD	Silicon Vertex Detector
SVN	Apache Subversion
UI	User Interface
UML	Unified Modeling Language
URD	User Requirements Definition

Bibliography

[1] M. Kobayashi and T. Maskawa, "CP Violation in the Renormalizable Theory of Weak Interaction," *Prog. Theor. Phys.*, vol. 49, pp. 652–657, 1973.

[2] A. Abashian *et al.*, "The Belle Detector," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 479, no. 1, pp. 117 – 232, 2002.

[3] Z. Doležal and S. Uno (editors), "Belle II Technical Design Report." KEK Report 2010-1, `arXiv:1011.0352v1`.

[4] M. Friedl *et al.*, "Readout and Data Processing Electronics for the Belle-II Silicon Vertex Detector," in *Topical Workshop on Electronics for Particle Physics*, (Paris, France), pp. 417–421, September 2009.

[5] M. J. French *et al.*, "Design and results from the APV25, a deep sub-micron CMOS front-end chip for the CMS tracker," *Nuclear Instruments and Methods in Physics Research A*, vol. 466, pp. 359–365, July 2001.

[6] C. Irmler *et al.*, "Electronics and mechanics for the Silicon Vertex Detector of the Belle II Experiment," in *Topical Workshop on Electronics for Particle Physics*, (Aachen, Germany), September 2010.

[7] L. Jones, *APV25-S1 User Guide*. Rutherford Appleton Laboratory, Harwell Oxford, Didcot OX11 0QX, September 2001.

[8] C. Irmler, "Upgrade Studies for the Belle Silicon Vertex Detector," Master's thesis, Vienna University of Technology, September 2008.

[9] M. Friedl *et al.*, "The Origami Chip-on-Sensor Concept for Low-Mass Readout of Double-Sided Silicon Detectors," in *Topical Workshop on Electronics for Particle Physics*, (Naxos, Greece), September 2008.

[10] H. Furtado *et al.*, "Delay 25 an ASIC for timing adjustment in LHC," in *11th Workshop on Electronics for LHC and future Experiments*, (Geneva), pp. 148–152, CERN, September 2005.

[11] CAEN S.p.A., *A2818 PCI Optical Link - Technical Information Manual*, 1st ed., August 2008.

[12] CAEN S.p.A., *A3818 PCI Express Optical Link - Technical Information Manual*, 3rd ed., April 2011.

[13] CAEN S.p.A., *V2718 VME PCI Optical Link Bridge - Technical Information Manual*, 9th ed., June 2009.

[14] Struck Innovative Systeme GmbH, *SIS1100(-e)/3104 PCI (Express) to VME interface*, 1.04 ed., February 2012.

[15] Austrian Academy of Sciences, Institute of High Energy Physics, *APVDAQ, APV25 Readout System, Reference Manual*, August 2007.

[16] National Instruments Corporation, "LabWindows®/CVI Website." `http://www.ni.com/lwcvi`, April 2012.

[17] M. Friedl *et al.*, "FADC+PROC Schematics." `http://www.hephy.at/project/electronic2/PDF/ARCHIV/BELLE/BELLE_NEU/FADC_PROC/SCH_fadc+proc.pdf`, February 2007.

[18] T. Obermayer, "HEPwiki: TuxDAQ Requirements (Restricted Access)." `http://www.hephy.at/wiki/index.php/TuxDAQ_Requirements`, 2011-2012.

[19] "IEEE Recommended Practice for Software Requirements Specifications," *IEEE Std 830-1998*, 1998.

[20] I. Sommerville, *Software Engineering*. Amsterdam: Addison-Wesley Longman Publishing Co., Inc., 9th revised edition ed., February 2010.

[21] Linux.org, "Linux Logos and Mascots." `http://www.linux.org/info/logos.html`, September 2012.

[22] P. Stevens and R. Pooley, *Using UML: Software Engineering with Objects and Components*. Pearson Education Ltd., updated edition ed., 1999, 2000.

[23] K. Beck and W. Cunningham, "A Laboratory for Teaching Object-Oriented Thinking," in *OOPSLA*, pp. 1–6, 1989.

[24] P. Eden, "A Step by Step Method for Conceptual Data Analysis," in *Software Engineering: Education and Practice, 1996. Proceedings. International Conference*, pp. 42 –49, January 1996.

[25] B. Lavoie *et al.*, "Customizable Descriptions of Object-Oriented Models," in *Proceedings of the 5th Conference on Applied Natural Language Processing (ANLP-97)*, March 1997.

[26] R. C. Martin, *Agile Software Development, Principles, Patterns and Practices*. Prentice-Hall, Inc., 1st ed., 2002.

[27] R. C. Martin, *Designing Object Oriented Applications using UML*. Prentice Hall, 2nd ed., 1999.

[28] R. C. Martin, "The Open-Closed Principle," *C++ Report*, January 1996.

[29] P. Deitel and H. Deitel, *C++ - How to Program*. Pearson Education Ltd., 7th ed., 2010.

[30] R. C. Martin, "The Liskov Substitution Principle," *C++ Report*, March 1996.

[31] R. C. Martin, "The Dependency Inversion Principle," *C++ Report*, May 1996.

[32] E. Gamma *et al.*, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[33] G. E. Krasner and S. T. Pope, "A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80," *J. Object Oriented Program.*, vol. 1, pp. 26–49, Aug. 1988.

[34] CERN, "ROOT - C++ Coding Conventions." `http://root.cern.ch/root/Conventions.html`, August 2012.

[35] B. Stroustrup, *The C++ Programming Language*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 3rd ed., 2000.

[36] CERN, "ROOT - List of Data Types." `http://root.cern.ch/root/html/ListOfTypes.html`, August 2012.

[37] D. van Heesch, "Doxygen." `http://www.doxygen.org`, August 2012.

[38] The Boost Community, "The Boost C++ Libraries." `http://www.boost.org`, August 2012.

[39] CERN, "The CERN ROOT Framework." `http://root.cern.ch`, August 2012.

[40] The Apache Software Foundation, "Apache log4cxx." `http://logging.apache.org/log4cxx/`, August 2012.

[41] The Apache Software Foundation, "Apache log4j." `http://logging.apache.org/log4j/`, August 2012.

[42] The Apache Software Foundation, "The Apache License." `http://www.apache.org/licenses/`, August 2012.

[43] JellyCan Code, "SimpleIni." `http://code.jellycan.com/simpleini/`, August 2012.

[44] Massachusetts Institute of Technology, "The MIT License." `http://copyfree.org/licenses/mit/license.txt`, August 2012.

[45] Nokia, "Qt - Cross-Platform Application and UI Framework." `http://qt.nokia.com`, August 2012.

[46] U. Rathmann, "Qwt - Qt Widgets for Technical Applications." `http://qwt.sourceforge.net`, August 2012.

[47] Free Software Foundation, Inc., "GNU Lesser General Public License (LGPL)." `http://www.gnu.org/copyleft/lesser.html`, August 2012.

[48] U. Frank, "Delegation: An Important Concept for the Appropriate Design of Object Models," *Journal of Object-Oriented Programming*, vol. 13, no. 3, pp. 13–18, 2000.

[49] J. Rumbaugh *et al.*, *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[50] Free Software Foundation, Inc., "GNU Make." `http://www.gnu.org/software/make/`, October 2012.

[51] LightySoft, "LogMX - The Universal Log File Viewer." `http://www.logmx.com/`, Oktober 2012.

[52] ASUSTeK Computer Inc., "ASUS P8H67-V." `http://www.asus.com/Motherboards/Intel_Socket_1155/P8H67V/`, October 2012.

[53] Software in the Public Interest, Inc. *et al.*, "Debian 6.0 - AMD64 Port." `http://www.debian.org/ports/amd64/`, October 2012.

[54] CERN, "Scientific Linux CERN 5." `http://linux.web.cern.ch/linux/scientific5/`, October 2012.

[55] Red Hat, Inc., "Red Hat Enterprise Linux." `http://www.redhat.com/products/enterprise-linux/`, October 2012.

[56] Free Software Foundation, Inc., "GCC, the GNU Compiler Collection." `http://gcc.gnu.org/`, October 2012.

[57] Free Software Foundation, Inc., "GDB: The GNU Project Debugger." `http://www.gnu.org/software/gdb/`, October 2012.

[58] The Apache Software Foundation, "Apache Subversion." `http://subversion.apache.org/`, August 2012.

[59] The Eclipse Foundation, "Eclipse IDE for C/C++ Developers." `http://www.eclipse.org/downloads/moreinfo/c.php2`, October 2012.

[60] T. Obermayer, "TuxDAQ Repository (Restricted Access)." `svn://heros.hephy.at/Belle2/tuxdaq`, April 2012.