National Institute of Applied Sciences and Technology

CARTHAGE UNIVERSITY

Graduation Project

Specialty : **Software Engineering**

Aare: a flexible data analysis library for hybrid pixel detectors

Presented by

Bechir Braham

INSAT Supervisor : **Dr. Guesmi Ghada**
Company Supervisor : **Dr. Froejdh Erik**

Presented on : **30/09/2024**

JURY

M. President FLEN (President)
Ms. Reviewer FLENA (Reviewer)

Academic Year : 2023/2024

Acknowledgements

Thank you all!

Table of Contents

List of Figures v

List of Tables vi

Abstract vii

General Introduction 1

I Project Context and Scope 3

1 Presentation of The Host Company . . . 3
2 Detector's Group Presentation and Work . . . 4
3 Problem Statement . . . 6
3.1 Existing Solution . . . 6
3.2 Limits of The Existing Solution . . . 6
3.2.1 Code Complexity . . . 6
3.2.2 Code Rigidity . . . 6
3.2.3 Code Duplication . . . 7
3.2.4 Data Storage Limitations . . . 7
4 Project Goals . . . 7
5 Work Methodology . . . 8
5.1 Tools . . . 8
5.2 Benefits of Kanban . . . 8
5.3 Meetings . . . 9
6 High Level Planning . . . 9

II Requirement Specification and Overall Architecture 12

1 Requirement Specification . . . 13
1.1 Actors Identification . . . 13
1.2 Functional Specification . . . 13
1.2.1 File Input/Output . . . 13
1.2.2 Network Input/Output . . . 13
1.2.3 Data Processing . . . 14
1.2.4 Python Interface . . . 14
1.3 Non-Functional Specification . . . 14

1.3.1 Performance . . . 14
1.3.2 Reliability . . . 14
1.3.3 Portability . . . 15
1.3.4 Extensibility . . . 15
1.3.5 Maintainability . . . 15
1.4 Requirements Specification Analysis . . . 15
2 Overall Architecture and Guidelines . . . 16
2.1 Clean Architecture . . . 17
2.2 SOLID Principles . . . 19
2.2.1 Single Responsibility Principle (SRP) . . . 19
2.2.2 Open/Closed Principle (OCP) . . . 19
2.2.3 Liskov Substitution Principle (LSP) . . . 19
2.2.4 Interface Segregation Principle (ISP) . . . 20
2.2.5 Dependency Inversion Principle (DIP) . . . 20
2.3 C++ Specific Design . . . 20
2.3.1 Templates Metaprogramming . . . 20
2.3.2 C++ Idioms . . . 22
2.3.3 C++ Limitations . . . 22
2.4 Project Architectural Design . . . 22
2.4.1 Project Modules . . . 22
2.4.2 Project Architecture . . . 22
2.4.3 Class Diagram . . . 22

III Implementation of a Flexible Data Analysis Library for Hybrid X-ray Particle Detectors **24**

1 Project Setup . . . 25
1.1 Project Structure . . . 25
1.2 Build System . . . 25
1.3 Testing and Continuous Integration . . . 25
2 Core Module Implementation . . . 25
3 File IO Module Implementation . . . 25
4 Network IO Module Implementation . . . 25
5 Processing Module Implementation . . . 25
6 Python Bindings Implementation . . . 25

IV Library Applications and Evaluation **27**

1 Examples of Library Applications 27

2 Parallelization and Multi-threading 27

3 Performance Evaluation 27

Conclusion and Perspectives **28**

Appendix : Miscellaneous remarks **31**

List of Figures

I.1 slsDetectorPackage setup for two detectors. Configuration uses TCP while data streaming from detector to slsReceiver uses UDP 5

I.2 Gantt Diagram of the 6 phases of development. A margin was left at the end of the project to account for holidays, vacation days and development delays 10

II.1 Diagram of the Clean Architecture: the circles represent the different layers of the architecture, the arrows represent the dependencies between the layers. The bottom right part of the diagram shows how dependencies are inverted. 18

List of Tables

Abstract

This is the english abstract of your project. It must be longer and presented in more details than the abstract you write on the back of your report.

General Introduction

La Table de matière est la première chose qu'un rapporteur va lire. Il faut qu'elle soit :

- Assez détaillée [1]. En général, 3 niveaux de numéros suffisent;
- Votre rapport doit être réparti en chapitres équilibrés, à part l'introduction et la conclusion, naturellement plus courts que les autres;
- Vos titres doivent être suffisamment personnalisés pour donner une idée sur votre travail. Éviter le : Conception , mais privilégier : Conception de l'application de gestion des ... Même s'ils vous paraissent longs, c'est mieux que d'avoir un sommaire impersonnel.

Une introduction doit être rédigée sous forme de paragraphes bien ficelés. Elle est normalement constituée de 4 grandes parties :

1. Le contexte de votre application : le domaine en général, par exemple le domaine du web, de BI, des logiciels de gestion ?
2. La problématique : quels sont les besoins qui, dans ce contexte là, nécessitent la réalisation de votre projet?
3. La contribution : expliquer assez brièvement en quoi consiste votre application, sans entrer dans les détails de réalisation. Ne pas oublier qu'une introduction est censée introduire le travail, pas le résumer;
4. La composition du rapport : les différents chapitres et leur composition. Il n'est pas nécessaire de numéroter ces parties, mais les mettre plutôt sous forme de paragraphes successifs bien liés.

[1] Sans l'être trop

Part I

Chapter 1

Chapter I

Project Context and Scope

Summary

1 Presentation of The Host Company 3

2 Detector's Group Presentation and Work 4

3 Problem Statement 6

3.1 Existing Solution 6

3.2 Limits of The Existing Solution 6

4 Project Goals 7

5 Work Methodology 8

5.1 Tools 8

5.2 Benefits of Kanban 8

5.3 Meetings 9

6 High Level Planning 9

Introduction

In this first chapter we will present the Paul Scherrer Institute, the host company of the project. We will also introduce

1 Presentation of The Host Company

The Paul Scherrer Institute (PSI) is the largest research institute for natural and engineering sciences within Switzerland. Created in 1998, the institute is located in Canton Aargau and employs around 2300 people. PSI is composed of 8 main research centers:

- Center for Life Sciences
- Center for Neutron and Muon Sciences
- Center for Nuclear Engineering and Sciences
- Center for Energy and Environmental Sciences
- Center for Photon Science
- Center for Scientific Computing
- Theory and Data
- Center for Accelerator Science and Engineering.

In addition the Paul Scherrer Institute is famous for its synchotron: the Swiss Light Source (SLS). The SLS is a third-generation synchrotron light source, which provides high-brilliance photon beams with high spectral resolution and tunable energy. It is used for a wide range of experiments in materials science, biology and chemistry. [Bög02; Web; Lv2222]

2 Detector's Group Presentation and Work

The Detector's Group is one of the research groups at the Paul Scherrer Institute. It is part of the Laboratory for X-ray Nanoscience and Technologies (LXN) which itself is part of the Center for Photon Science.
The group is responsible for the development of new detectors, the maintenance of the existing ones, and the development of software for the data acquisition and analysis. It is responsible for the development of new technologies for the detectors, such as new sensors, readout electronics, and data acquisition systems. The group is also involved in the development of new techniques for the data analysis, such as image processing, pattern recognition, and machine learning. These Detector's are an integral part of the beamline's setup, and are used to detect the x-ray photons that are emitted by the synchrotron. Many experiments in different fields can use these detectors such as for studying the properties of materials [But+24], protein structures [Pom+09], crystallography [Leo+23], biology [Lem+23; Dul+24], and many other fields of research.
On the software side, the group is responsible for the development of the software that is used to control the detectors, acquire the data, and analyze it. The software is developed in C++ and is used by the scientists to perform their experiments. The main package maintained by

the detector's group is the "SLS Detector Package" available publicly on https://github.com/slsdetectorgroup/slsDetectorPackage. The package provides several binaries such as:

- **slsReceiver** The receiver server acquires incoming data from detectors using UDP and listens for configurations from host machine using TCP.
- **slsDetectorGet** Used by the host machine to request the configuration on the Receiver or the Detector.
- **slsDetectorPut** Used to configure parameters on both the Receiver and Detector.
- **slsDetectorGui** A graphical user interface (GUI) that receives data from the Receiver and displays it.

The list of binaries is not exhaustive, and the package contains many other binaries for simulating detectors, analyzing data, and many other utilities.

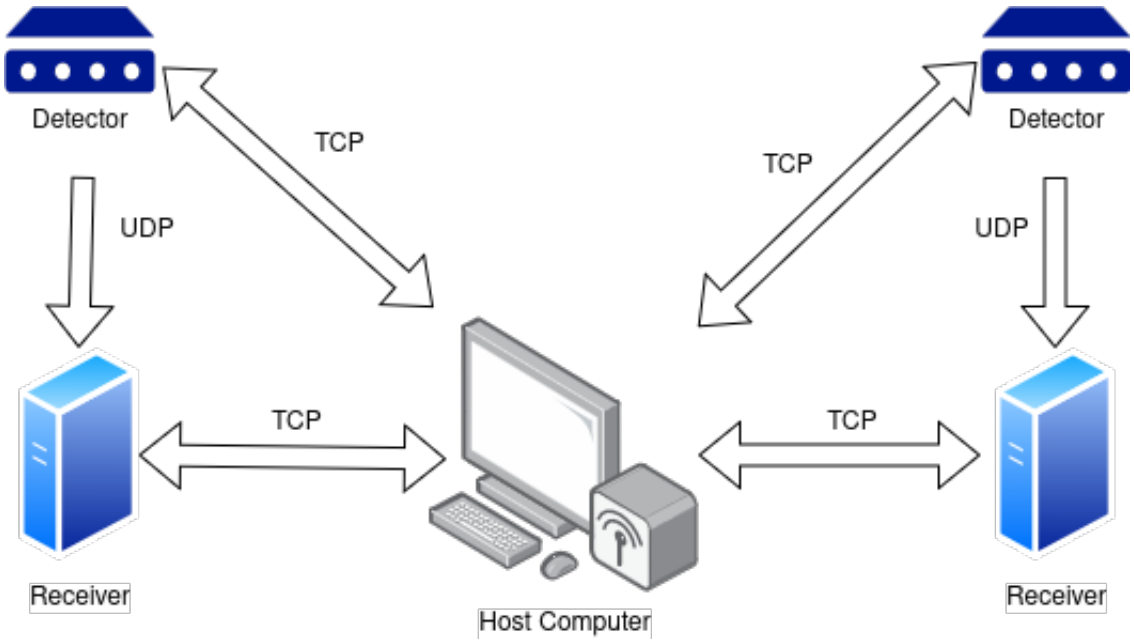

Figure I.1 – slsDetectorPackage setup for two detectors. Configuration uses TCP while data streaming from detector to slsReceiver uses UDP

3 Problem Statement

3.1 Existing Solution

The detector's group includes scientists, software engineers, firmware engineers, chip designers and many more roles. The group is diverse and the libraries' usage differs from one user to another. In general the usage of the libraries includes: acquiring data from network , configuring receivers and detectors, storing incoming data, processing data on the fly or after storing it. For the standard functionalities users rely on the slsDetectorPackage binaries. But the slsDetectorPackage is a generic software developed for public use and has very broad functionalities. Hence, for specific use cases scientists might need to write their own scripts or change the slsDetectorPackage source code and build again.

3.2 Limits of The Existing Solution

The slsDetectorPackage is a very powerful software package, but it has some major limitations.

3.2.1 Code Complexity

First, the code is very complex and has a steep learning curve. The code is written in C++ and uses many advanced features of the language. The code is also very large, with many thousands of lines of code. This makes it difficult for new users to understand how the code works, and to modify it to suit their needs. In addition, scientists are not software engineers, and in case of a bug, new feature or a specific use case they will be exposed to complex code that they are not familiar with. This might include the need to understand C++ code, multi-threading, network programming, and many other advanced topics.

3.2.2 Code Rigidity

Second, the code is very rigid and inflexible. The code is designed to work in a specific way, and it is difficult to modify it to work in a different way. This means that scientists are limited in what they can do with the code, and they are forced to work within the constraints of the existing code. This can be very frustrating for scientists, who may have specific requirements that are not met by the existing code. Furthermore, some of the speicific use case code is very brittle and can break easily if the code is modified. It lacks proper testing, error handling and logging. This makes it difficult to maintain and extend the code.

3.2.3 Code Duplication

Third, the code is duplicated in many places. Scientist often rely on their own scripts to perform specific tasks. This leads to each scientist having their own version of the code, which is difficult to maintain and update. and also results in the use of sub-optimal code, which is not efficient or reliable.

3.2.4 Data Storage Limitations

As the detectors become more and more advanced, the amount of data that they produce is increasing. This has made processing the incoming data in real-time a must. The slsDetector-Package provides very limited functionalities for processing data on the fly. This means that scientists are forced to store the data on disk and process it later. This is not ideal and is becoming less practical as the amount of data can be very expensive to store and process. The new library should be designed to process around 10GB/s of data in real-time.

4 Project Goals

The goal of this project is to develop a new library that will address the limitations of the existing software. The new software package will be designed to be simple, flexible, and efficient. It will be easy to use, and will be designed to meet the needs of scientists and engineers.

The library should include functionalities for acquiring data from receiver servers, stream data to receivers, read and write raw data files and numpy files, includes commonly used algorithms for data processing and it should expose a C++ and a Python API.

In addition, code should be well tested, documented, and should include logging and error handling. The library should be designed to be extensible, so that new features can be added easily in the future. and also flexible so that it accomodates different and upcoming use cases.

The library should be designed to be efficient, so that it can process data in real-time, and should be able to handle large amounts of data. It should use parallelism to distribute the load on multiple cores, and should be able to take advantage of the GPU for processing data. On the other hand it should abstract the low level details of the hardware and network communication to make it easy to use for the scientists.

5 Work Methodology

We used the **Kanban** methodology to manage the project. Kanban is a subsystem of the Toyota Production System (TPS), which was created in 1940s to control inventory levels, the production and supply of components, and in some cases, raw material. [JG10]
The board is divided into several columns:

- **Backlog** Contains all the tasks that need to be done.
- **To Do** Contains the tasks that are ready to be worked on.
- **In Progress** Contains the tasks that are currently being worked on.
- **Done** Contains the tasks that are completed.

5.1 Tools

We used Github Projects to manage the Kanban board. Github Projects is a tool that allows you to create a Kanban board and manage your tasks. It is integrated with Github, so you can link your tasks to your code, and track your progress easily. We also used Github Issues to create tasks, and Github Pull Requests to review and merge the code.
The Kanban boards were available for all the group members (involved in project or not), so that they can see the progress of the project, and contribute to it if needed. The boards were updated regularly, and the progress was tracked using the boards. The boards were also used to plan the work, and to assign tasks to the group members.

5.2 Benefits of Kanban

The Kanban methodology has several benefits:

- **Visibility** The Kanban board provides a visual representation of the work that needs to be done, and the progress that has been made.
- **Flexibility** The Kanban board is flexible, and can be easily adapted to the needs of the project.
- **Efficiency** The Kanban board helps to prioritize the work, and to focus on the most important tasks.
- **Collaboration** The Kanban board is a collaborative tool, and can be used by all the group members to track the progress of the project.

5.3 Meetings

We had regular meetings with the group members to discuss the progress of the project, and to plan the work. On each Tuesday, The whole group meets for about an hour to discuss the progress of the multiple projects that are being worked on. This meeting helps to keep everyone informed about the progress of the projects, and to identify any issues that need to be addressed.
In addition, on each Friday, a one-on-one meeting is held with the project supervisor to discuss the progress made during the week, and to plan the work for the next week. This is a more detailed meeting where we discuss the tasks that need to be done, and the kep track of the progress of the project.
Furthermore, the group has an open door policy, where anyone can ask for help, or discuss any issues that they are facing. Knowledge sharing is encouraged, and regular short discussions help overcoming the roadblocks that one might encounter.

6 High Level Planning

Even though the work methodology is highly flexible, we have a high level planning that we follow. The project is divided into several phases:

- **Phase 1: Research and Project Setup** In this phase, we researched the existing solutions, identified the limitations of the existing software, setup the project, and developed a high level architecture for the new library.
- **Phase 2: Implementation of the File Module** In this phase, we implemented the file IO module, which is responsible for reading and writing raw data files and numpy files.
- **Phase 3: Implementation of the Network Module** In this phase, we implemented the network module, which is responsible for acquiring data from receiver servers, and streaming data.
- **Phase 4: Implementation of the Processing Module** In this phase, we implemented the processing module, which includes commonly used algorithms for data processing.
- **Phase 5: Implementation of the Python API** In this phase, we implemented the Python API, which allows users to use the library from Python.
- **Phase 6: Documentation, Testing and Evaluation** In this phase, we tested the library more thoroughly, evaluated the performance, documented the code and added tutorials.

It is important to note that the phases are not fixed, and can be adapted to the needs of the project. The phases are used to plan the work, and to track the progress of the project. For example the python API was implemented in parallel with the other modules, as it was very useful to test the library and to get feedback from the users.

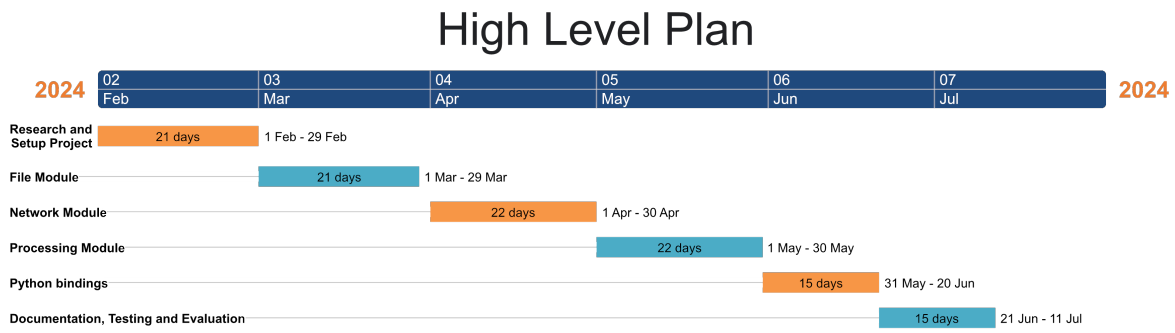

Figure I.2 – Gantt Diagram of the 6 phases of development. A margin was left at the end of the project to account for holidays, vacation days and development delays

Conclusion

La conclusion est en général sans numérotation, et n'apparaît pas dans la table des matières.

Part II

Chapter 2

Chapter II

Requirement Specification and Overall Architecture

Summary

1 Requirement Specification 13

1.1 Actors Identification 13

1.2 Functional Specification 13

1.3 Non-Functional Specification 14

1.4 Requirements Specification Analysis 15

2 Overall Architecture and Guidelines 16

2.1 Clean Architecture 17

2.2 SOLID Principles 19

2.3 C++ Specific Design 20

2.4 Project Architectural Design 22

Introduction

The requirement specification is the first step in the software development process. It is the basis for the design and implementation of the software system. It is the process of defining, documenting and maintaining the requirements of the system. The requirements are the description of the system services and constraints that are to be implemented.
In this chapter we will present the requirement specification of the project, we will define our actors, the functional and non-functional requirements of the system. We will also present the overall architecture of the project and the design principles that we will follow.

1 Requirement Specification

1.1 Actors Identification

Our library is developed for one main actor: the scientist.
The scientist is the person who is going to use the library to develop new data processing algorithms, acquire data from different IO sources, analyze the data and visualize the results. It is assumed that the scientist is very competent but is not necessarily an expert in software development. The library is designed to be easy to use and to provide a high level of abstraction to the scientist.

1.2 Functional Specification

1.2.1 File Input/Output

The library should provide the scientists an interface to handle file input/output:

- Read/Write frame data from different file formats: Raw, JSON, HDF5 and Numpy.
- Reorder the binary frame data according to the detector type.
- In case of part files (data split accross multiple files), the library should be able to read all the files and reconstruct the data.
- Read/Write metadata from different file formats: JSON, Raw.
- Define a new cluster file format (.clust2) to store processed cluster data. The file format must be flexible to accomodate different types of structured data.
- Read/Write from the new cluster file format.

1.2.2 Network Input/Output

The library should provide the scientists an interface to handle network input/output:

- Read frame data from multiple receivers using ZeroMQ. [Hin13]
- Syncronize the data streams coming from different receivers and merge them into a single data stream.
- Send data to multiple server using ZeroMQ. Thses servers can be other existing GUI applications.
- Provide an interface to distribute tasks accross multiple workers using ZeroMQ.

1.2.3 Data Processing

Although the task of data processing is very specific to the scientist, the library should provide a set of basic data processing algorithms that can be used as building blocks for more complex algorithms. In addition implementing these basic algorithms will help the scientist to understand the library and how to use it. Therefore helps in acclerating the adoption of the library.
Specifically the library should implement a Cluster Finder algorithm that can be used to find clusters in a frame of data. and also a pedestal subtraction algorithm that can be used to remove the background noise from the data.

1.2.4 Python Interface

Most scientists prefer to use Python for their data analysis tasks. Python has become the de facto language for data analysis and machine learning. It is easier to use than C/C++ and it less verbose. It has a large number of libraries that can be used for data analysis and visualizations.
Therefore the library should provide a Python interface that can be used to access the library functionalities. The Python interface should be easy to use, similar to other Python libraries like Numpy and Pandas.

1.3 Non-Functional Specification

1.3.1 Performance

The library is expected to handle around 10GB/s of data streaming from the network and process it in real time. In addition processing large data files should be done as fast as possible to allow the scientist to iterate quickly on their work. These are very challenging requirements and will require a lot of optimization in all levels of the library. The library should be able to run on a single CPU core and also on multiple CPU cores to parallelize the processing.

1.3.2 Reliability

The library should be able to handle different types of errors that can occur during the data processing. It should be able to recover from these errors and continue processing the data. The library should also be able to handle different types of data corruptions that can occur during the data acquisition. For example if a frame of data is corrupted (in a file or from the network), it should be skipped the program continues processing the rest of the data. Or in the case of a network error, the software should reconnect to the network and continue receiving the

data. Furthermore, Thorough testing on multiple levels and on different scenarios is required to ensure the integrity of the code.

1.3.3 Portability

Nothing is assumed about the platform on which the library will run. It must be able to run on different operating systems (Windows, Linux, MacOS). and on different hardware architectures (x86, ARM). It cannot rely on specific kernel features or syscalls that are not available on all platforms.

1.3.4 Extensibility

Use cases and requirements can change over time. The library should be able to adapt to these changes. The software design must allow developers and scientists to add new functionalities easily. Careful considerations must be taken to make easy to add features without breaking the existing code.

1.3.5 Maintainability

The development of the library is expected to continue for a long time. The library should be easy to maintain and evolve. The code should be well documented and easy to read. Explanations should be provided for complex algorithms and data structures. Helpful documents should be provided to help new developers understand the code and how to contribute to the project.

1.4 Requirements Specification Analysis

First, given the harsh performance requirements, the library should be implemented in a compiled language. Rust, C++, Java are possible candidates for this task. Rust is a new language that is gaining popularity for systems programming. It is designed to be safe and fast [BA22]. However, it is still a young language and the ecosystem is not as mature as C++. Java is a good candidate for performance and portability, but it is not as fast as C++. Therefore, C++ is the best candidate for this task. It is a mature language with a large ecosystem and a large number of libraries that can be used. It is also a very fast language that can be optimized to run close to the hardware. It is also a language that is well known by the scientist community.

Second, to provide high performance python code, the library should be implemented in C++ and then wrapped in Python using the Pybind11 library [JRM17]. Pybind11 is a lightweight

header-only library that exposes C++ types in Python and vice versa, mainly to create Python bindings of existing C++ code. It is easy to use and provides a high level of abstraction to the developer. It is also very fast and does not introduce a lot of overhead. It is also compatible with the C++ standard library and can be used with other C++ libraries.

Third, it is possible to modularize the library into different modules where each module is responsible for a specific domain. This will help in organizing the code and make it easier to maintain and evolve. It will also help in parallelizing the development of the library. The library can be divided into the following modules:

- core module: responsible for the basic data structures that will be used in all other modules.
- file_io module: responsible for reading and writing data from different file formats.
- network_io module: responsible for reading and writing data from the network.
- processing module: responsible for implementing the data processing algorithms.
- python module: responsible for exposing the library to Python.

Fourth, for more flexibility and extensibility, the library should be implemented using the Clean Architecture and the SOLID principles.
Finally, extensive testing is required to ensure the reliability of the library. Unit tests, integration tests and end to end tests are all required. In addition to that, the library should be tested on different platforms and different hardware architectures to ensure its portability.

2 Overall Architecture and Guidelines

Given the requirements of the project, software architecture and design principles must be carefully chosen to ensure the success of the project. In this section we will present some of the design principles and architectural patterns that we will follow in the project.

A software solution should provide to stakeholders two main things: behavior and structure. The behavior is the functionality of the system, what the system does. It answers for the functional and non-functional requirements of the system. The structure is the organization of the system, how the components of the system are organized and how they interact with each other. A good structure is essential for the maintainability and extensibility of the system.

Investing time in the design of the software architecture is very important. Specifications can change, requirements can change, but the architecture is the foundation of the system and it should be solid and flexible enough to accommodate these changes.

> The goal of software architecture is to minimize the human resources required to build and maintain the required system
>
> Robert C. Martin [Martin17]

As the quote by "Uncle Bob" suggests, the goal of software architecture is to reduce the software development costs in the project's early life. But also it is crucial to bring down the cost of new features and maintenance as project evolves and becomes more complex.

Several architectural patterns exist such as: Layered Architecture, Hexagonal Architecture [FP09], Clean Architecture [Martin17], Onion Architecture [Palermo08], DCI [CR14] etc. Although these patterns differ they share some common principles such as separation of concerns, testability, and independence from frameworks, databases, and UIs.

In this project we will follow the Clean Architecture.

2.1 Clean Architecture

The clean architecture was first introduced by Robert C. Martin (also known as Uncle Bob) in his blog post [MarBlog12] and later in his book "Clean Architecture: A Craftsman's Guide to Software Structure and Design" [Martin17]. Like other architectural patterns, the clean architecture promotes separation of concerns, testability, and independence from frameworks, databases, and UIs.
As shown in figure II.1, the clean architecture is composed of several layers:

- Entities: This layer contains the business objects of the system. These objects are the core of the system and they are independent of any other layer.
- Use Cases: This layer contains the application-specific business rules. It contains the use cases of the system. Each use case is a single action that the system can perform.
- Interface Adapters: This layer contains the interfaces that the system uses to communicate with the outside world. It contains the presenters, controllers, and gateways.

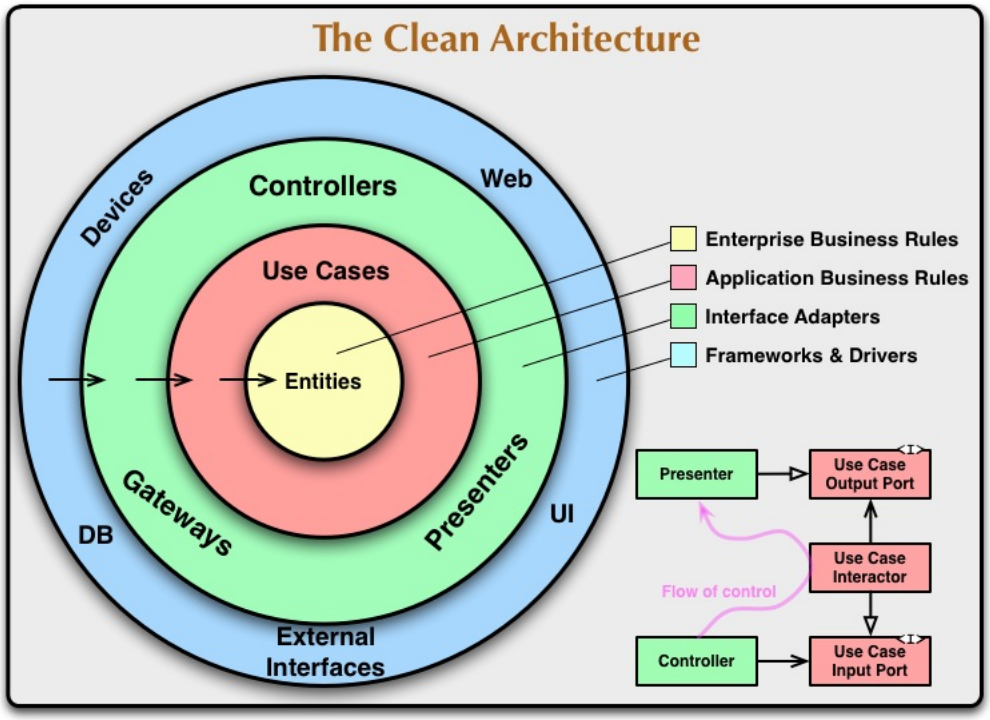

Figure II.1 – Diagram of the Clean Architecture: the circles represent the different layers of the architecture, the arrows represent the dependencies between the layers. The bottom right part of the diagram shows how dependencies are inverted.

- Frameworks and Drivers: This layer contains the frameworks and drivers that the system uses to communicate with the outside world. It contains the web frameworks, databases, and other external systems.

The clean architecture promotes the separation of concerns and the independence of the layers. Each layer is independent of the other layers and can be replaced without affecting the other layers. It is very important that the inner circles do not depend on the outer circles. The inner circles will not know anything about functions, classes, or modules in the outer circles. This known as the **Dependency Rule**.

> "This rule says that source code dependencies can only point inwards. Nothing in an inner circle can know anything at all about something in an outer circle." [MarBlog12]

For communication between the layers, interfaces are used. The inner layers define interfaces that the outer layers must implement or call. The dependency inversion principle (will be explained in the next section) is used to decouple the layers and make them independent of each other.

2.2 SOLID Principles

The SOLID principles are a set of five principles that are used to design object-oriented software. They were introduced by Robert C. Martin in his paper "Design Principles and Design Patterns" [Mar00].

2.2.1 Single Responsibility Principle (SRP)

The SRP states that:

> "A module should have only one reason to change." [Martin17]

This means that a class should have only one responsibility and serves one actor. If a class has more than one responsibility, it should be split into multiple classes. This will make the code easier to understand and maintain.

2.2.2 Open/Closed Principle (OCP)

The OCP states that:

> "A software artifact should be open for extension but closed for modification." [Mey97]

This means that the code should be designed in such a way that its features can be extended without the need to modify the existing code. This can be acheived by a plethora of design patterns and object oriented principles.

2.2.3 Liskov Substitution Principle (LSP)

As Barbara Liskov defined it in 1988:

> "If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T" [Lis87]

Or in simpler terms: Subtypes must be substitutable for their base types. In the case of classes this means that a subclass should be able to replace its parent class without affecting the correctness or behavior of the program.

2.2.4 Interface Segregation Principle (ISP)

The ISP states that:

> "Clients should not be forced to depend on methods that they do not use." [Mar03]

This means that interfaces should be small and focused. The ISP combats the problem of interface pollution. When classes depend on an interface it does not fully use, it is forced to implement or depend on methods that are not needed. This can lead to unnecessary dependencies and coupling between classes. and code might break when the interface is updated.
The solution proposed by the ISP is to split the interfaces into smaller interfaces that are more focused. This way the classes can depend on the interfaces that they need and not on the "fat" interface.

2.2.5 Dependency Inversion Principle (DIP)

The DIP is defined in the Agile Software Development book by Robert C. Martin [Mar03] as:

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

The dependency inversion principle is crucial for the successful implementation of the clean architecture. It is used to decouple the layers and make them independent of each other. The DIP is the mechanism that allows the inner layers to define interfaces that the outer layers must implement or call. This way the inner layers do not depend on the outer layers and can be replaced without affecting the other layers.

2.3 C++ Specific Design

The library will be implemented in C++ and will be wrapped in Python. For this reason some specific design considerations must be taken into account. C++ is a very powerful language that provides a lot of features that can be used to implement almost any design. However, it is also a very complex language that has its own pitfalls and limitations.

2.3.1 Templates Metaprogramming

C++ provides a powerful feature called templates that can be used to implement metaprogramming. Metaprogramming is the writing of programs that write or manipulate other programs.

Templates allows the programmer to write generic code that will be instantiated at compile time. This can be used to implement generic algorithms that can work with different types of data. Templates can make the code much more efficient as it moves the computation from runtime to compile time.

Templates are a core design feature of C++. They are used extensively in the C++ standard library and in many other libraries. However, templates can be very complex and can lead to very long compile times and cryptic compile error messages.

From a software design perspective, being aware of templates and how to use them is very important. Templates can be used for static polymorphism, to implement generic algorithms, and to implement compile-time checks.

The very important constraint to keep in mind for this project is that the C++ library will be wrapped in Python. In practice this is done by shared libraries that are loaded by the Python interpreter. Hence, the C++ code must be compiled first before it can be used in Python.

Listing II.1 – Example of a Templated Function

```
template <typename T, int n>
T add(T a) {
    return a+n;
}
```

In the above example II.1, the function add is a templated function that takes a type T and an integer n as template parameters. The function adds n to the input parameter a and returns the result.

In C++ code whenever we call this function with a different set of parameters the compiler will generate a new version of the function. However, in Python we cannot do this as the interpreter will not be able to generate new versions of the function at runtime.

Therefore, the use of templates in the C++ code must be limited to the minimum and only used when it is really necessary. The C++ code should be as generic as possible but not at the expense of the Python interface.

In some use cases we are able to instantiate the C++ templated code for a set of types or values that the function expects. For example we can define function for numerical types (int8, int16,

uint8, uint16, float, double ...) However we should always be aware that this can lead to code bloat and increase the size of the shared library.

2.3.2 C++ Idioms

2.3.3 C++ Limitations

2.4 Project Architectural Design

2.4.1 Project Modules

2.4.2 Project Architecture

2.4.3 Class Diagram

Conclusion

Faire ici une petite récapitulation du chapitre, ainsi qu'une introduction du chapitre suivant.

Part III

Chapter 3

Chapter III

Implementation of a Flexible Data Analysis Library for Hybrid X-ray Particle Detectors

Summary

1 Project Setup 25

1.1 Project Structure 25

1.2 Build System 25

1.3 Testing and Continuous Integration 25

2 Core Module Implementation 25

3 File IO Module Implementation 25

4 Network IO Module Implementation 25

5 Processing Module Implementation 25

6 Python Bindings Implementation 25

Introduction

1 Project Setup

1.1 Project Structure

1.2 Build System

1.3 Testing and Continuous Integration

2 Core Module Implementation

3 File IO Module Implementation

4 Network IO Module Implementation

5 Processing Module Implementation

6 Python Bindings Implementation

Conclusion

Part IV

Chapter 4

Chapter IV

Library Applications and Evaluation

Summary

1 Examples of Library Applications **27**

2 Parallelization and Multi-threading **27**

3 Performance Evaluation **27**

Introduction

1 Examples of Library Applications

2 Parallelization and Multi-threading

3 Performance Evaluation

Conclusion

Conclusion and Perspectives

Bibliography

[BA22] William Bugden and Ayman Alahmar. “Rust: The programming language for safety and performance”. In: *arXiv preprint arXiv:2206.05503* (2022).

[Bög02] M Böge. “First operation of the swiss light source”. In: *Proc. EPAC*. Vol. 2. 2002.

[But+24] Tim A Butcher et al. “Ptychographic nanoscale imaging of the magnetoelectric coupling in freestanding BiFeO3”. In: *Advanced Materials* (2024), p. 2311157.

[CR14] James O Coplien and Trygve Reenskaug. “The dci paradigm: Taking object orientation into the architecture world”. In: *Agile Software Architecture*. Elsevier, 2014, pp. 25–62.

[Dul+24] Christian Dullin et al. “In vivo low-dose phase-contrast CT for quantification of functional and anatomical alterations in lungs of an experimental allergic airway disease mouse model”. In: *Frontiers in medicine* 11 (2024), p. 1338846.

[FP09] Steve Freeman and Nat Pryce. *Growing object-oriented software, guided by tests*. Pearson Education, 2009.

[Hin13] Pieter Hintjens. *ZeroMQ: messaging for many applications*. ” O’Reilly Media, Inc.”, 2013.

[JG10] Muris Lage Junior and Moacir Godinho Filho. “Variations of the kanban system: Literature review and classification”. In: *International journal of production economics* 125.1 (2010), pp. 13–21.

[JRM17] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. *pybind11 – Seamless operability between C++11 and Python*. https://github.com/pybind/pybind11. 2017.

[Lem+23] Tali Lemcoff et al. “Brilliant whiteness in shrimp from ultra-thin layers of birefringent nanospheres”. In: *Nature Photonics* 17.6 (2023), pp. 485–493.

[Leo+23] Filip Leonarski et al. “Kilohertz serial crystallography with the JUNGFRAU detector at a fourth-generation synchrotron source”. In: *IUCrJ* 10.6 (2023).

[Lis87] Barbara Liskov. “Keynote address-data abstraction and hierarchy”. In: *Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*. 1987, pp. 17–34.

[Lv2222] Q. Z. Lv et al. “High-Brilliance Ultranarrow-Band X Rays via Electron Radiation in Colliding Laser Pulses”. In: *Phys. Rev. Lett.* 128 (2 Jan. 2022), p. 024801. DOI: 10.1103/PhysRevLett.128.024801. URL: https://link.aps.org/doi/10.1103/PhysRevLett.128.024801.

[Mar00] Robert C Martin. “Design principles and design patterns”. In: *Object Mentor* 1.34 (2000), p. 597.

[Mar03] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. USA: Prentice Hall PTR, 2003. ISBN: 0135974445.

[MarBlog12] Robert C. Martin. "The Clean Architecture". In: (2012). URL: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html.

[Martin17] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. 1st. USA: Prentice Hall Press, 2017. ISBN: 0134494164.

[Mey97] Bertrand Meyer. *Object-oriented software construction (2nd ed.)* USA: Prentice-Hall, Inc., 1997. ISBN: 0136291554.

[Palermo08] Jeffrey Palermo. "The Onion Architecture". In: (2008). URL: https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/.

[Pom+09] Daniel A Pomeranz Krummel et al. "Crystal structure of human spliceosomal U1 snRNP at 5.5 Å resolution". In: *Nature* 458.7237 (2009), pp. 475–480.

[Web] PSI Website. *About Swiss Light Source SLS*. URL: https://www.psi.ch/en/sls/about-sls.

Appendix : Miscellaneous remarks

- Un rapport doit toujours être bien numéroté;
- De préférence, ne pas utiliser plus que deux couleurs, ni un caractère fantaisiste;
- Essayer de toujours garder votre rapport sobre et professionnel;
- Ne jamais utiliser de je ni de on, mais toujours le nous (même si tu as tout fait tout seul);
- Si on n'a pas de paragraphe 1.2, ne pas mettre de 1.1;
- TOUJOURS, TOUJOURS faire relire votre rapport à quelqu'un d'autre (de préférence qui n'est pas du domaine) pour vous corriger les fautes d'orthographe et de français;
- Toujours valoriser votre travail : votre contribution doit être bien claire et mise en évidence;
- Dans chaque chapitre, on doit trouver une introduction et une conclusion;
- Ayez toujours un fil conducteur dans votre rapport. Il faut que le lecteur suive un raisonnement bien clair, et trouve la relation entre les différentes parties;
- Il faut toujours que les abréviations soient définies au moins la première fois où elles sont utilisées. Si vous en avez beaucoup, utilisez un glossaire.
- Vous avez tendance, en décrivant l'environnement matériel, à parler de votre ordinateur, sur lequel vous avez développé : ceci est inutile. Dans cette partie, on ne cite que le matériel qui a une influence sur votre application. Que vous l'ayez développé sur Windows Vista ou sur Ubuntu n'a aucune importance;
- Ne jamais mettre de titres en fin de page;
- Essayer toujours d'utiliser des termes français, et éviter l'anglicisme. Si certains termes sont plus connus en anglais, donner leur équivalent en français la première fois que vous les utilisez, puis utilisez le mot anglais, mais en italique;
- Éviter les phrases trop longues : clair et concis, c'est la règle générale !

Rappelez vous que votre rapport est le visage de votre travail : un mauvais rapport peut éclipser de l'excellent travail. Alors prêtez-y l'attention nécessaire.

APPENDIX : MISCELLANEOUS REMARKS

33