CODEN:LUTEDX/(TEIE-3158)/1-119(2026)

Industrial Electrical Engineering and Automation

# A Framework for Modular PLC Programming Based on Object Oriented Design



**Jonathan Edenburg**

**Oliver Simonsson**

Division of Industrial Electrical Engineering and Automation
Faculty of Engineering, Lund University

# A Framework for Modular PLC Programming Based on Object Oriented Design


LUND UNIVERSITY

LTH at Campus Helsingborg

Department of Industrial Electrical Engineering and Automation

2026

Bachelor thesis:

Jonathan Edenburg

Oliver Simonsson

LTH at Campus Helsingborg

Lund University

Box 882

SE-251 08 Helsingborg

Sweden

LTH vid Campus Helsingborg

Lunds universitet

Box 882

251 08 Helsingborg

Printed in Sweden

Lunds universitet

Lund

# ABSTRACT

This thesis investigates the development of an object-oriented modular framework for PLC systems, intended for use at Etteplan, an international engineering consultancy. The thesis was motivated by recurring challenges in PLC projects, including unclear program flows, low code reusability, and limited scalability. These issues justify the need for a standardized framework grounded in the IEC 61131-3 standard, fourth edition. Using design science research as the overarching methodology, the thesis combines a literature review and source code review with semi-structured interviews with practicing automation engineers. The result is a platform-independent framework design based on modular architecture and object-oriented constructs, accompanied by development guidelines. The framework template was partially implemented in Beckhoff's development platform TwinCAT 3. The framework aims to improve code reuse, maintainability, and knowledge transfer within Etteplan's projects.

**Keywords**: *IEC 61131-3, PLC programming, modular architecture, object-oriented programming, industrial automation, Design Science Research*

# SAMMANFATTNING

Detta examensarbete undersöker utvecklingen av ett modulärt, objektorienterat ramverk för PLC-system, avsett för användning på Etteplan, ett internationellt teknikkonsultföretag. Examensarbetet utgick från återkommande utmaningar i PLC-projekt, däribland otydliga programflöden, låg kodåteranvändbarhet och begränsad skalbarhet. Denna problematik motiverar behovet av ett standardiserat ramverk grundat i standarden IEC 61131-3 fjärde upplaga. Med designvetenskaplig metodik, kombinerar examensarbetet en litteratur och källkods-översikt med semistrukturerade intervjuer med verksamma automationsingenjörer. Resultatet är en plattformsoberoende ramverksdesign baserad på modulär arkitektur och objektorienterade principer, åtföljd av utvecklingsriktlinjer. Ramverkets mall implementerades delvis i Beckhoffs utvecklingsplattform TwinCAT 3. Ramverket syftar till att förbättra kodåteranvändning, underhållbarhet och kunskapsöverföring inom Etteplans projekt.

**Keywords**: *IEC 61131-3, PLC-programmering, modulär arkitektur, objektorienterad programmering, industriell automation*

# PREFACE

First and foremost, we would like to express our sincere gratitude to our industry supervisor, Marcus Bou at Etteplan, for his invaluable guidance, support, and expertise at every stage of this thesis. We are also deeply grateful to Fredrik Svensson and everyone at Etteplan's Lomma office for giving us the opportunity to conduct our research within their organization, and for providing the essential resources and knowledge that made this work possible.

Furthermore, we would like to extend our appreciation to our academic supervisor, Morten Hemmingsson at Lund University. His expertise was instrumental in shaping both the direction and the overall quality of this report.

Completing this thesis has been a profound learning experience for both of us, offering invaluable real-world insight into PLC development and the practical challenges of applying software engineering principles in an industrial setting. We hope that the work presented here proves valuable to Etteplan and to others working in the field of automation engineering.

Helsingborg, May 2026
Jonathan Edenburg & Oliver Simonsson

# Contents

# List of Figures

# Abbreviations

| | |
|---|---|
| AI | Artificial Intelligence |
| DSR | Design Science Research |
| EQM | Equipment Module |
| FB | Function Block |
| FUN | Function |
| HMI | Human-Machine Interface |
| IEC | International Electrotechnical Commission |
| ISA | International Society of Automation |
| LD | Ladder Diagram |
| OOP | Object-Oriented Programming |
| PLC | Programmable Logic Controller |
| POU | Program Organization Unit |
| PRG | Program |
| ST | Structured Text |
| TIA | Totally Integrated Automation |
| UML | Unified Modeling Language |

# 1. INTRODUCTION

In industrial automation, the use of control systems is widespread. To realize a control system, programmable logic controllers (abbreviated as PLC) are used. These systems are used to control machinery and various types of processes within production and manufacturing environments. PLC systems are often developed for a certain type of machine or production line, and are configured in accordance with customer requirements. This can result in projects being developed by different engineers with varying programming styles, structures, and documentation practices. Furthermore, this may make it difficult to obtain a comprehensive understanding of existing programs that were originally written by one engineer or a team of engineers and subsequently maintained by others in the future.

This chapter provides the context and motivation for this thesis. The background is presented in Section 1.1, introducing Etteplan and the purpose behind the thesis. The purpose and goals are defined in Sections 1.2 and 1.3, followed by problem formulation and research questions in Section 1.4. The scope and limitations are stated in Section 1.6, and the division of work is outlined in Section 1.7.

This report is organized as follows.

- **Chapter 2** presents the technical background, including the IEC 61131-3 standard for PLC software architecture, with the focus on the fourth edition's addition of object-oriented programming. It also covers UML modeling in a PLC context and describes the industrial context of six core components as identified by Etteplan.
- **Chapter 3** describes the methodologies used and their application across the five work-phases of the thesis.
- **Chapter 4** presents the analysis in three parts:
  - Thematic analysis of the interviews conducted
  - Application of coding practices for the framework, as a general architecture with a focus on object-oriented programming
  - Evaluation of the six core components, based on identified design decisions

- **Chapter 5** presents the results for each core component, using UML diagrams and code examples showing how the template is used.

## 1.1 Etteplan

Etteplan is an international consultancy company that offers engineering services within, among other areas, industrial automation. The company is involved in the development and commissioning of automation solutions for the manufacturing industry, where the programming of automated systems constitutes a central component of many projects. Within the field of automation, Etteplan develops and implements machine and production line control systems, encompassing the design of control logic, hardware integration, as well as commissioning and on-site service.

The automation engineers at Etteplan have identified recurring challenges in the execution of their PLC projects. They contend that extensive use of global variables can make program flows unclear by causing dependencies within the program, which could further lead to a reduction of modularity and limited functionality for simulation and testing. This is attributed to the absence of a standardized program architecture that is consistently reused across their various projects. As a result, similar functionality is reimplemented for each project, for example, the handling of alarms, states, recipes, and communication. The consequence is low reusability and limited scalability, which in turn complicates maintenance and further development. Furthermore, the lack of structure could also restrict junior engineers from learning by researching existing codebases.

Modern industrial automation makes extensive use of the IEC 61131-3 standard, which defines programming languages for PLC systems. The standard encompasses, among other things, standardized data types and program structure that enable modular programming through constructs such as function blocks. It also facilitates the implementation of object-oriented concepts (International Electrotechnical Commission [IEC], 2025). Nevertheless, in practice, only a limited subset of the standard's capabilities are typically utilized in real-world projects.

Etteplan has proposed a potential solution in the form of a general framework for PLC

programs. The framework should consist of a platform-independent template based on a modular architecture, object-oriented structure, and accompanied by the corresponding documentation and development guidelines.

The proposed template should consist of a modular design based on core architectural components, deemed by Etteplan to be generally universal for all machines. The most critical identified core components, in order of priority, are:

- Machine-state coordination
- Alarm & Event handling
- Recipe handling
- Safety program integration
- Stop & Recovery handling
- Input/Output handling

Each of these core components encapsulates a distinct concern of the control system of a machine. The template's purpose is therefore to provide a consistent and maintainable starting point to be extended with application-specific logic.

## 1.2 Purpose

The purpose of this thesis is to develop, in collaboration with Etteplan, a standardized modular template for PLC programs that Etteplan can apply in future automation projects, to improve code quality, readability, and reusability, and serve as an introduction for junior engineers.

## 1.3 Goals

The framework shall aim to support relevant PLC programming languages with a focus on Structured Text, and a concrete implementation shall be developed in Beckhoff's development environment, TwinCAT 3 (Beckhoff Automation, 2024), demonstrating how the framework can be applied. The work shall also provide documentation in the form of UML diagrams and version control guidelines to ensure the framework can be maintained and extended within the organization.

The specific objectives of the thesis are to:

- Develop a general PLC program template based on Etteplan's specification.
- Implement a functioning template project mainly in TwinCAT 3.
- Create example equipment modules demonstrating the framework in use.
- Produce documentation and UML diagrams for the template project.
- Evaluate the solution through simulation or against available hardware where possible.

## 1.4 Problem Formulation

The absence of a standardized structure for PLC programs leads to several interconnected problems. Programs become difficult to understand for engineers encountering them for the first time, resulting in extended onboarding time and increased risk of errors during modification. Troubleshooting and servicing are similarly complicated, as logic is often distributed across many program blocks without a clear overarching structure.

A further consequence is that opportunities for code reusability are limited, since programs are typically tightly coupled to specific projects or machines. Similar functionality is therefore implemented multiple times in different ways, increasing development time and reducing the long-term quality and maintainability of the systems.

The following research questions will be addressed in the thesis:

- How are PLC programs typically structured in the industry today?
- What is required for the framework to be platform-independent?
- What problems arise during maintenance and troubleshooting?
- How can a general PLC architecture be designed for scalability and reusability?
- How can object-oriented principles be applied to PLC programming in accordance with IEC 61131-3?
- How should core components be managed in a standardized framework?
- How can the solutions be tested and simulated?

## 1.5 Motivation of the Thesis

This thesis bridges theoretical OOP principles with industrial PLC architecture, addressing a practical need in automation engineering. For Etteplan, the thesis can establish a shared, reusable PLC structure that is applicable both to customer projects and to the onboarding of new engineers. A standardized framework with well-defined core components reduces the effects of differing programming styles and supports more consistent working practices throughout the organization. This can reduce startup times, reduce initial effort in new projects, and improve troubleshooting and maintenance. More broadly, well-structured industrial software reduces the risk of operational downtime and safety incidents, as a clearer program structure lowers the likelihood of errors during both development and maintenance.

## 1.6 Limitations

The primary implementation is limited to TwinCAT 3. Support for other platforms, such as TIA Portal or Studio 5000, is considered at the architectural level but is not implemented within the scope of this thesis.

## 1.7 Division of Work

The work of this thesis was divided equally between the two authors across all phases, as summarized in Table 1.

Table 1: Division of work distribution in percentage (%).

| **Area** | **Jonathan Edenburg** | **Oliver Simonsson** |
|---|---|---|
| Analysis | 50 | 50 |
| Design | 50 | 50 |
| Implementation | 50 | 50 |
| Report | 50 | 50 |
| Presentation | 50 | 50 |

# 2. TECHNICAL BACKGROUND

This chapter presents the theoretical background relevant to the thesis. Section 2.1 introduces the IEC 61131-3 standard. Section 2.2 covers the classic PLC programming constructs. Section 2.3 discusses the OOP extensions introduced in the 2025 edition. Section 2.4 outlines the PLC execution model. Section 2.5 introduces UML in a PLC context. Section 2.6 describes the core architectural components of PLC programs.

## 2.1 The IEC 61131-3 Standard

IEC 61131-3 is an international standard for PLCs that defines a software architecture and a set of programming languages for the automation industry. It provides predefined data types and structures to support modular programming, and, as of the latest revision, enables OOP. The standard encompasses both textual and graphical languages. This thesis focuses primarily on Structured Text (ST) and briefly addresses Ladder Diagram (LD).

### *2.1.1 Structured Text (ST)*

ST is a text-based, high-level language derived from Pascal. It uses structured control elements such as IF-THEN-ELSE statements and loops. Its readability makes it the preferred language in modern PLC development (Antonsen, 2020). Code Example 2.1 shows a basic ST if-statement.

```
1 //Basic motor control in Structured Text
2 IF StartButton AND NOT Alarm THEN
3     MotorOn := TRUE;
4 ELSE
5     MotorOn := FALSE;
6 END_IF;
```

Code Example 2.1 – Structured text example.

### *2.1.2 Ladder Diagram (LD)*

Ladder Diagram is a graphical language that originated from the physical representation of electrical logic circuits, organized to represent the logical flow of energy from left to right.

LD uses visual symbols to represent input or conditions. Due to its similarity to traditional electrical schematics, LD remains widely used in the industry (Antonsen, 2020). The ladder diagram representation of the logic from code example 2.1 can be seen in Figure 2.1.

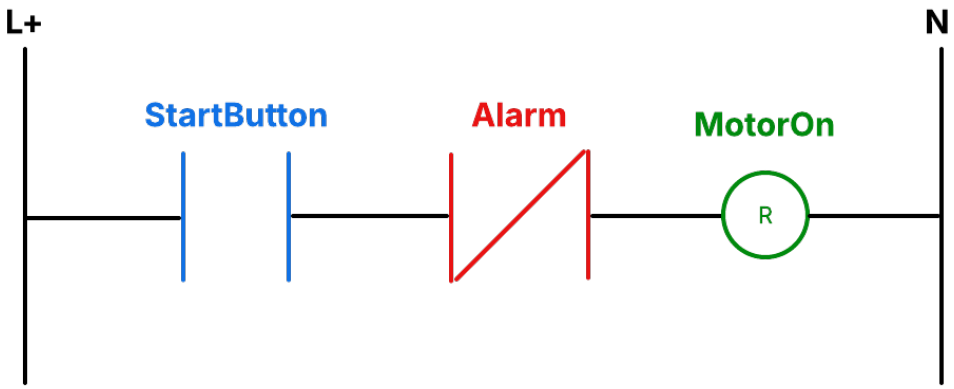


Figure 2.1: Ladder diagram of basic motor control.

## 2.2 Program Organization Unit – Classic IEC

Program Organization Units (POUs) are modular and self-contained blocks of code that are used in PLC programming. The IEC-61131-3 standard defines POUs as the smallest independent software units of a user program. Three types of POUs are supported by the standard: Program (PRG), Function Block (FB), and Function (FUN) (IEC, 2013; John and Tiegelkamp, 2010), and they can be implemented in any of the languages defined in the IEC 61131-3 standard (IEC, 2025).

POUs allow modular programming design by dividing automation logic into smaller units with predefined responsibilities. Developers are able to reuse POUs within the same project by calling instantiated POUs (IEC, 2013). This can improve program structure, code readability and maintainability if used in the intended way (John and Tiegelkamp, 2010).

Each POU manages and maintains its own internal variables locally. This makes internal variables inaccessible from outside the scope of the POU. Instead, input and output variables are defined for interfacing with a POU, providing controlled access to its functionality (IEC, 2013). In general, POUs can be called by other POUs, with some key behavioral differences between the three types (John and Tiegelkamp, 2010). POUs are composed of two parts: the

variable declaration and implementation section. The variable declaration has delimiters, which tell the compiler how to interpret what kind of variables are declared (IEC, 2025). The most commonly used delimiters are listed in Table 2. Furthermore, the implementation section starts after 'END_VAR' and ends with END_PROGRAM, corresponding to each POU (IEC, 2013).

Table 2: Variable Declaration types.

| Delimiter | Purpose |
|---|---|
| VAR | Local/Internal variables. |
| VAR_INPUT | Variables passed by value from the caller. |
| VAR_OUTPUT | Variables passed back to the caller. |
| VAR_IN_OUT | Variables passed by reference must be assigned in the call. |
| VAR_GLOBAL | Accessible across whole project,<br>declared separately, no associated POU. |
| VAR_CONSTANT | Variable value that cannot be changed during runtime. |

POUs are generally hardware independent as they implement and describe abstractions of automation control logic. This further enhances their reusability, as they can be reused on different PLC platforms, sometimes with minimal modifications (John and Tiegelkamp, 2010).

### *2.2.1 Programs*

Programs (PRG) are a type of POU. They can be defined with input and output variables, while also maintaining internal state variables. Within a PRG, instantiations of FBs can be declared, as well as calls to FUNs. PRGs are a single instance object in a PLC project, which means that a PRG is called by directly referencing its name. A PRG can be assigned to a PLC task, which is executed by the runtime system, usually called the Main PRG. This acts as the entry point of the program (IEC, 2013), similar to the main method in Java programming (Oracle Corporation, 2026).

### *2.2.2 Function Blocks*

Function blocks (FBs) are a more versatile POU compared to programs. They represent reusable code segments that accept defined input values and can produce output values, while

instantiated FBs maintain internal state variables between execution cycles (IEC, 2013). FBs are typically used for logic that takes advantage of internal state memory, such as timers, counters, or state logic (John and Tiegelkamp, 2010). Multiple instances of FBs may be created and accessed using an instance variable, each with its own internal state. FBs may call other FBs and FUNs within their code logic segment (IEC, 2013). This allows for similar behavior as an instance of an object from a class in other object-oriented programming languages such as Java (John and Tiegelkamp, 2010).

### *2.2.3 Functions*

Functions are similar to FBs, but they have no internal state variables that persist between executions. Instead, they represent stateless logical units that can produce deterministic outputs based on the input values when called. This means calling a FUN, given the same input, will always produce the same output (IEC, 2013). Therefore, FUNs are typically used for deterministic calculations such as mathematical operations and data manipulation, for example, string manipulations or unit conversions (John and Tiegelkamp, 2010). FUNs can call other FUNs but not FB or PRGs (IEC, 2013).

### *2.2.4 Data Types*

Beyond the constructs described in the preceding sections, IEC 61131-3 (IEC, 2013) defines a set of user-defined data types that allow engineers to express structured and constrained data within the type system.

#### *2.2.4.1 Enumerations*

An enumeration defines a named type whose valid values are restricted to a finite, explicitly declared set of identifiers. Rather than representing states or categories as raw integers, an enumeration assigns a meaningful name to each value, making the intention of a variable self-documenting and allowing the compiler to reject any assignment that falls outside of the declared set (IEC, 2013). A variable typed as an enumeration cannot take an arbitrary numeric value, which eliminates a class of errors that arise when numeric literals are used to represent categorical data (John and Tiegelkamp, 2010).

#### *2.2.4.2 Structs*

A struct is a composite data type that groups a fixed set of named fields into a single unit. Each field carries its own type, and the struct as a whole can be declared as a variable, passed as a function block input or output, or stored in an array (IEC, 2013). Structs are commonly used to bundle related signals or parameters that logically belong together, allowing them to be managed and passed as a cohesive unit rather than as a collection of independent variables (IEC, 2013; John and Tiegelkamp, 2010).

#### *2.2.4.3 Global Variable List*

IEC 61131-3 defines the Global Variable List (GVL) as a mechanism for declaring variables that are accessible across the entire PLC project. Variables declared in a GVL are not scoped to any particular POU and can be read or written by any PRG, FB, or FUN within the project (IEC, 2013).

GVLs are commonly used for mapping input and output, where physical hardware addresses are assigned to named variables that can then be referenced throughout the program, and for constants of configuration values that must be accessible from multiple locations. When used for this purpose, they provide a single traceable point of definition for values that are shared across the system (IEC, 2013; John and Tiegelkamp, 2010).

## 2.3 Newer Edition IEC 61131-3

The second edition of the IEC 61131-3 focused on procedural and modular programming using POUs (IEC, 2003). The third edition extended this foundation by introducing object-oriented programming (OOP) constructs, including inheritance, interface and methods. With the support for these constructs, the development of PLC systems was able to be structured according to modern software engineering principles, while keeping compatibility with the IEC programming standards (IEC, 2013). The fourth edition further refined these constructs by supporting principles such as encapsulation, abstraction, polymorphism and composition (IEC, 2025). The implementation of these principles is discussed in 2.3.1 – 2.3.6.

### 2.3.1 *Encapsulation*

Encapsulation is the concept that an object bundles and hides its data and internal logic which operates on that data. Interaction between the object and external code is then only possible through what the object exposes through public interfaces (Booch, 2007). This principle is reflected in the latest edition of IEC 61131-3, where FBs act as object-oriented components. Each FB instance maintains its own internal state and executes independently, allowing it to model physical devices or logical subsystems such as motors, valves, or sensors (IEC, 2025). Encapsulation in this context is achieved by restricted access to internal variables and exposing only the necessary interface (inputs, outputs, or methods) (IEC, 2025), seen in Code Example 2.2. This separation can reduce dependencies between modules and can improve maintainability.

```
1  FUNCTION_BLOCK FB_ConveyorMotor
2  VAR   // Private - only accessible inside this FB
3      MotorRunning : BOOL;
4  END_VAR
5  VAR_INPUT // Input variables passed by value
6      MotorSpeed: REAL;
7  END_VAR
8  VAR_OUTPUT // Public - visible to any code using this FB
9      Fault : BOOL; // Logic of how a fault is detected is
          ↪ internal exposing if there is a detected fault only
10 END_VAR
11 VAR_IN_OUT // Input variables passed by reference, meaning
      ↪ they can be written to within the POU
12 END_VAR
```

Code Example 2.2 – Encapsulation in FBs.

### 2.3.2 *Methods*

FBs can be declared to have methods, that define operations associated with a specific object. Methods have access to the internal variables of their object. They also provide a structured way of organizing logic within an FB into smaller, reusable units of code. Methods can be declared with two types of modifier, which can alter how they are accessed and their behavior (IEC, 2025). Table 3 lists these modifiers.

Table 3: Method modifiers.

| Modifier | Description |
|---|---|
| PUBLIC | Accessible from the outside of the declaring object. |
| PRIVATE | Accessible only inside the declaring object. |
| PROTECTED | Accessible in declaring the object and derived objects. |
| ABSTRACT | Method to be overridden. |
| FINAL | Method cannot be overridden. |
| OVERRIDE | Overrides the inherited method. |

Implementation of a method is shown in Code Example 2.3, tied to the conveyor motor FB in Code Example 2.2. This supports the principle of encapsulation, as methods allow for controlled access to internal variables, while preventing direct external access (IEC, 2025).

```
1  METHOD Start : BOOL
2  VAR
3  END_VAR
4  //Variable declaration associated with the method
5  IF Fault THEN
6      MotorRunning := FALSE; //This attribute is owned by the FB
7      Start := FALSE; //The name of the method is the return
       ↪ value
8  ELSE
9      MotorRunning := TRUE;
10     MotorSpeed := 2;
11     Start := TRUE;
12 END_IF;
13 END_METHOD
```

Code Example 2.3 – Start Method of the ConveyorMotor FB.

### *2.3.3 Polymorphism Realized Through Interfaces*

In the 2025 revision of the IEC 61131-3 standard, interfaces have been refined and extended from previous editions. Interfaces do not contain any implementation, but instead define a set of methods and properties that any FB that implements the interface must abide by. FBs that implement the same interface provide identical external functionality and can therefore be used interchangeably. This allows for software modules to communicate through defined interfaces without depending on a specific implementation (IEC, 2025). Code example 2.4

shows a conveyor motor interface, contracting for three methods.

```
INTERFACE I_ConveyorMotor
    METHOD Start : BOOL
    METHOD Stop  : BOOL
    METHOD Reset
//All methods must be implemented by FBs that implement this
   ↪ interface
```

Code Example 2.4 – Conveyor motor interface.

Polymorphism is the ability of code to operate on different object types through their common implemented interface, where the correct behavior is determined at runtime (Booch, 2007). Polymorphism is realized by interfaces within the PLC-programming environment and the IEC 61131-3 standard. A specific implementation of an FB can be accessed through a common interface, while providing different implementations for that interface. The calling program only interacts with the interface definition and does not know which specific FB type is used. With this, it is possible to substitute different implementations of an FB without modifying the surrounding program logic (IEC, 2025). For example, hardware with the same functionality, from different manufacturers, may implement the same interface and therefore be interchangeable within the system (John and Tiegelkamp, 2010). This concept is shown in Code Example 2.5, where the sorting station FB has a variable of type I_ConveyorMotor, which can be instantiated as any FB that implements I_ConveyorMotor. When the method 'Start' is called, the corresponding method is called from the FB, which is represented as the interface ConveyorMotor.

```
// Conveyor motor of a specific type or brand
FUNCTION_BLOCK FB_ConveyorMotorTypeOne IMPLEMENTS I_ConveyorMotor
    METHOD Start : BOOL
        IF Fault THEN
            MotorRunning := FALSE;
            Start := FALSE;
            RETURN;
        ELSE
            MotorRunning := TRUE;
            MotorSpeed := 5; //Specific speed for MotorTypeOne
            Start := TRUE;
            //Further logic specific to starting MotorTypeOne
        END_IF;
    END_METHOD
END_FUNCTION_BLOCK

//Conveyor motor of another type or brand
FUNCTION_BLOCK FB_ConveyorMotorTypeTwo IMPLEMENTS I_ConveyorMotor
    METHOD Start : BOOL
        IF Fault THEN
            MotorRunning := FALSE;
            Start := FALSE;
            RETURN;
        ELSE
            MotorRunning := TRUE;
            MotorSpeed := 10; //Specific speed for MotorTypeTwo
            Start := TRUE;
            //Further logic specific to starting MotorTypeTwo
        END_IF;

    END_METHOD
END_FUNCTION_BLOCK
//Now the machine module that uses a conveyor can use it without knowing how it works internally.
FUNCTION_BLOCK FB_SortingStation
VAR
    ConveyorMotor : I_ConveyorMotor;   // Either of the two motors above can be  instantiated into variable ' ConveyorMotor'
END_VAR
METHOD Run
    Motor.Start();   // Does not change regardless of which motor is used because of the common interface
END_METHOD

//...Further logic for a sorting station.
```

Code Example 2.5 – Polymorphism of conveyor motors in a sorting station.

### *2.3.4 Inheritance*

Inheritance means that objects can express "is a" relationships amongst each other. This allows an FB to be derived from another FB while extending or modifying its functionality. A derived FB inherits variables and methods from the base FB and may extend it by adding new functionality. Inheritance allows common functionality to be implemented once in a base FB and reused many times in an extended FB, by extending the base FB (IEC, 2025). Code example 2.6 shows implementation using inheritance to describe the commonality of methods and attributes between conveyor motors. As well as how the conveyor motor 'type three' inherits functionality from the base. Further, it shows the "is a" relationship and how it implements the unique start-method and variable 'MaxRPM'.

```
FUNCTION_BLOCK ABSTRACT FB_ConveyorMotorBase
VAR //All ConveyorMotors have at least these variables
    MotorRunning : BOOL;
END_VAR
VAR_INPUT
    MotorSpeed : REAL;
END_VAR
VAR_OUTPUT
    Fault      : BOOL;
END_VAR
//Further variable declaration
METHOD ABSTRACT Start
  //Specific ConveyorMotorType logic, to be overridden
END_METHOD
METHOD Stop : BOOL
    MotorRunning := FALSE;
    Stop         := TRUE;
END_METHOD
//All Conveyor Motors Stop and Reset work the same
METHOD Reset
    Fault      := FALSE;
    MotorRunning := FALSE;
END_METHOD

FUNCTION_BLOCK FB_ConveyorMotorTypeThree EXTENDS
   ↪ FB_ConveyorMotorBase
VAR
    MaxRPM : REAL;
END_VAR

METHOD Start : BOOL //METHOD Start is now overridden by the
   ↪ derived FB
    IF MotorSpeed > MaxRPM THEN  // not configured for safe
       ↪ operation
        Fault := TRUE;       // Fault inherited from
           ↪ FB_ConveyorMotorBase
        Start := FALSE;
        RETURN;
    END_IF;
    MotorRunning := TRUE;  // MotorRunning inherited from
       ↪ FB_ConveyorMotorBase
    Start      := TRUE;
    //Further ConveyorMotorTypeThree logic here
END_METHOD

// Methods Stop and Reset are inherited - no need to override
   ↪ them
```

Code Example 2.6 – Inheritance in PLC-programming.

### 2.3.5 *Composition*

Composition is the OOP principle allowing objects to express a "has a" relationship to each other. Therefore, a complex object could be defined by the sum of its simpler parts (Booch, 2007). In a PLC context, composition can be done by the declaration of instances of FBs within another FB. With this hierarchy, the outer FB delegates logic responsibility to the internal instances of contained FBs (IEC, 2025). An example of composition is seen in code example 2.7. The sorting station, FB_SortingStation, instantiates an FB_ConveyorMotor, allowing the logic for the motor to be delegated to that FB. It also instantiates other FBs, necessary for a sorting station.

```
1  FUNCTION_BLOCK FB_SortingStation
2  VAR
3      ConveyorMotor    : FB_ConveyorMotor; //Delegates Motor-
          ↪ logic to a Motor  FB
4      Scanner : FB_BarcodeScanner;//Delegates scanning-logic to
          ↪ Barcode Scanner FB
5      Pusher: FB_Pusher; // Delegates box-pushing logic to a
          ↪ Pushing FB
6  END_VAR
7  METHOD Run
8      IF Scanner.ReadLabel() = 'LANE_1' THEN
9          ConveyorMotor.Stop();
10         Pusher.Push();
11         //Further logic for the sorting here...
12     END_IF;
13 END_METHOD
```

Code Example 2.7 – Composition within a sorting station FB.

### 2.3.6 *Limitations of OOP Within PLC Architecture*

Although OOP offers significant advantages in industrial automation, it is not universally appropriate for every part of a PLC program. Simple operations such as direct input and output mapping, where the abstraction introduced by OOP would add complexity without meaningful benefit. Real-time requirements can also limit the application of OOP, though the concern is more specific than a general incompatibility. The risk lies in particular implementation patterns such as deep inheritance hierarchies and excessive use of polymorphism

in time-critical paths. Inattentive use of these patterns can introduce unpredictability in execution time (Cruz Salazar and Vogel-Heuser, 2020). This places additional demands on the developer and requires disciplined design choices.

## 2.4 Program Execution and Input/Output Handling

The execution of a program within a PLC differs significantly from traditional computer software. While a standard PC application may run asynchronously or be interrupted by background processes, a PLC operates using a deterministic execution model (Antonsen, 2020).

### *2.4.1 The Cyclic Scan*

To ensure that the control system responds to changes in the physical environment at predictable intervals, a PLC executes its logic in a continuous loop. According to Antonsen (2020), this is the core of PLC operation and is known as the Cyclic Scan.

The Cyclic Scan consists of three primary phases: reading input, executing program logic and writing outputs. This cycle is essential for enabling the system to be monitored in a deterministic way and thus ensure the system remains responsive and stable (Antonsen, 2020).

### *2.4.2 Reading Inputs*

During the first phase of the cyclic scan, all connected sensors, switches and components on the machine supply the PLC with values via the hardware input modules. This step also involves updating the data communication link, where variables are received from units such as control panels or other control systems (Antonsen, 2020). This ensures the internal program acts on the most current data possible before any logic is processed.

### *2.4.3 Executing Program Logic*

The next phase is where the PLC executes its instructions. This is done exactly once per scan and this includes program modules and FBs previously mentioned in section 2.3. As stated by Antonsen (2020), the PLC operates in real-time by executing program modules at fixed time intervals; this requires these modules to be processed within a very short, defined

time frame.

### *2.4.4 Writing Outputs*

Once fully processed, the resulting values from the program logic are written to output modules. This phase sets physical signals to components such as motors, valves and instruments based on the logic determined in the previous step. In addition to this, data is sent back via the communication link to other units. Once this final phase is completed, the cycle immediately repeats on the next rising edge of the scan time, starting back at the reading phase again to begin a new program scan (Antonsen, 2020).

### *2.4.5 I/O Processing*

Instead of reading physical inputs directly every time a variable is accessed in the code, the PLC takes a snapshot of all inputs at the beginning of the cycle and stores them in memory. This concept is called Process Image (IEC, 2003) and ensures consistency during the execution of the program.

Similarly, output commands are not sent to the hardware immediately as the code runs and are only updated physically after the program has finished its scan. This is, as noted by Antonsen (2020), done to prevent logic errors that could occur if an input changed state in the middle of a program scan.

## 2.5 Unified Modeling Language

The Unified Modeling Language (UML) is a standardized modeling language with the purpose of providing a common, platform-agnostic, visual representation for specifying and documenting the architecture of software systems (Object Management Group, 2017). UML defines fourteen diagram types divided into structural diagrams, which capture the static composition of the system, and behavioral diagrams, which capture its dynamic aspects (Object Management Group, 2017). This thesis will make use of three of these types: class diagrams, state machine diagrams and sequence diagrams.

### *2.5.1 Class Diagrams*

Class diagrams are structural diagrams that describe the building blocks of a system, including the types it defines, the relationships between them and the contracts they expose. In the context of the PLC, a class diagram shows the Function Blocks (FBs), their attributes, methods, inheritance and interfaces (Object Management Group, 2017). With the use of the previous code examples 2.7, the UML class diagram for the example system is represented in Figure 2.2, showing the relations, attributes, and components. It also specifies how the relationships between objects are illustrated using different arrows, and the contents of an object.

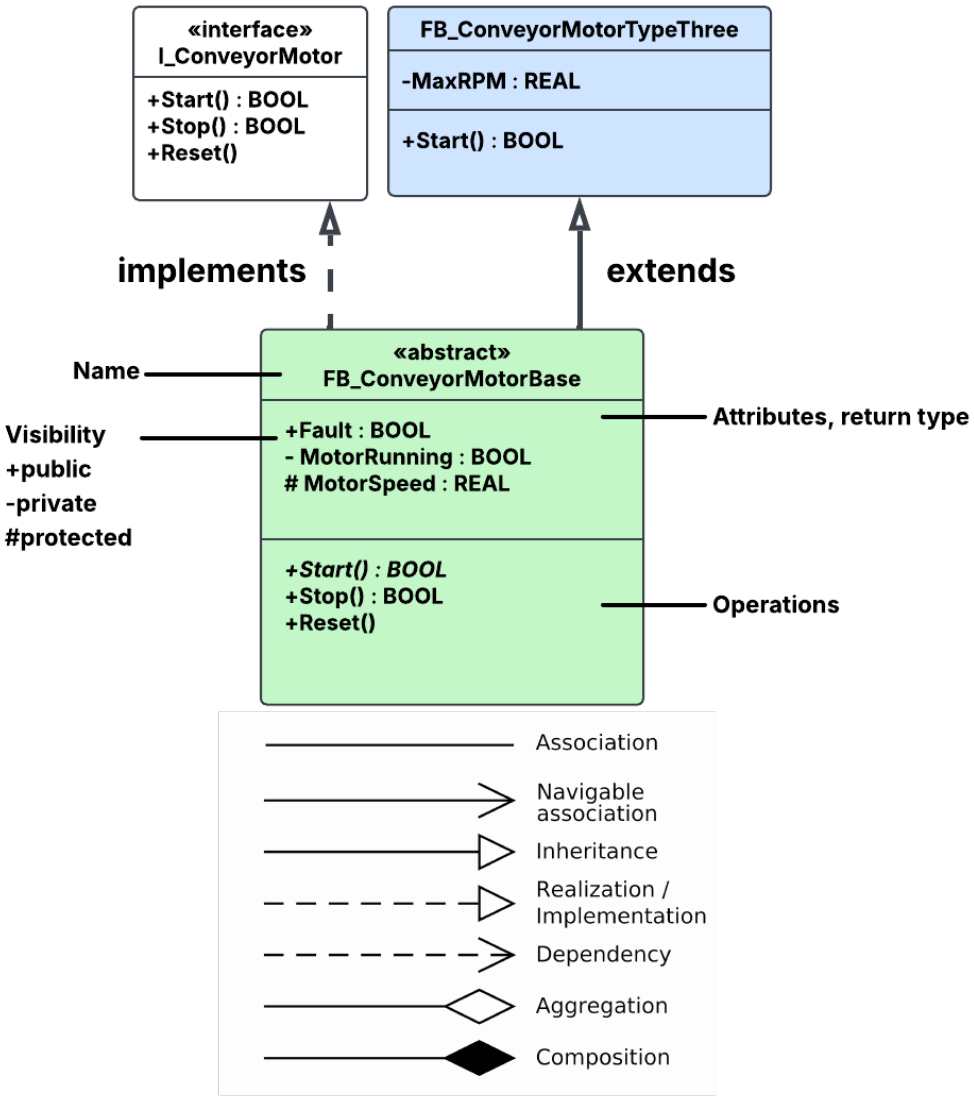


Figure 2.2: UML Class diagram example.

### *2.5.2 State Machine Diagrams*

The state machine diagrams are behavioral diagrams that illustrate the discrete states a system or object can have, the events that trigger transitions between them and any condition associated with those transitions (Object Management Group, 2017). This is illustrated in Figure 2.3, using the previous sorting station example with the introduction of a state machine. Each box represents a state, and the contents 'do:' correspond to logic within that state. The arrows pointing out from a state constitute the condition for transitioning to another state. A starting state is illustrated at the top, represented by the black circle. The

transition from the starting state has no condition and occurs immediately.

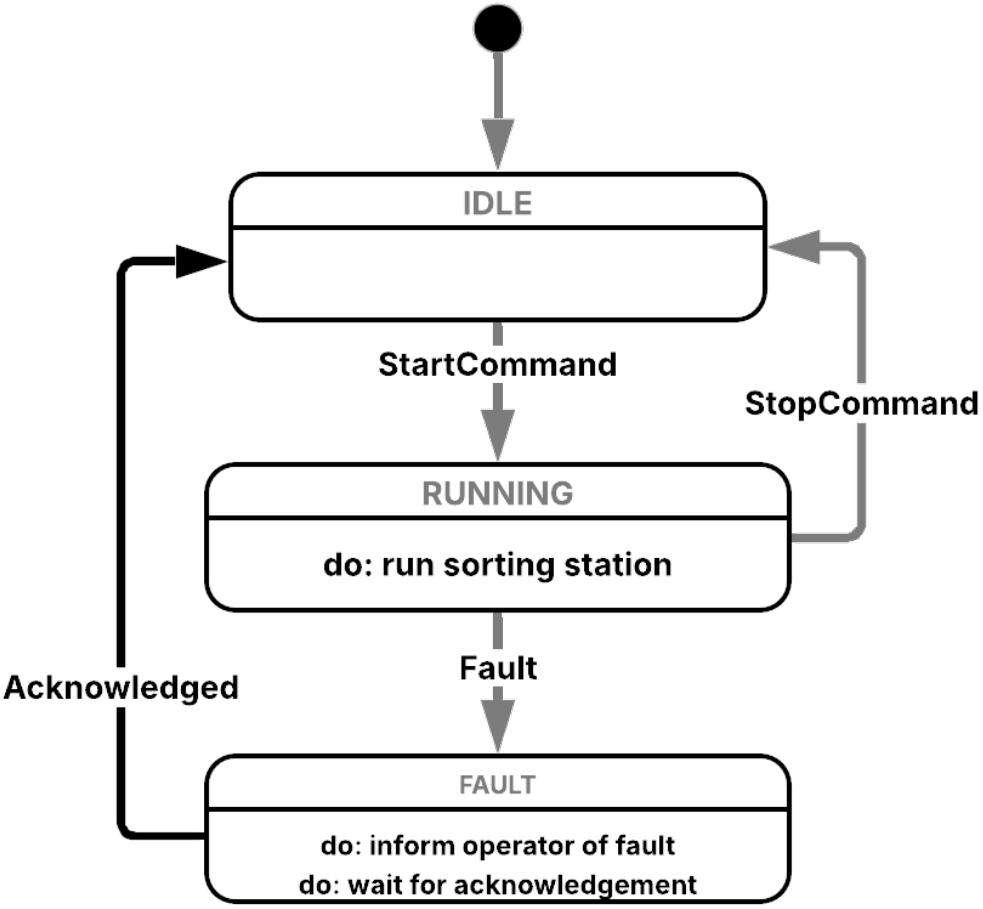


Figure 2.3: UML State diagram example.

### *2.5.3 Sequence Diagrams*

Sequence Diagrams are behavioral diagrams that show how objects interact over time, representing communication exchanged in chronological order along vertical lifelines (Object Management Group, 2017). A sequence diagram for the sorting stations run method from the previous code example 2.7 is illustrated in Figure 2.4. Showing the interaction between the sorting station and its objects.

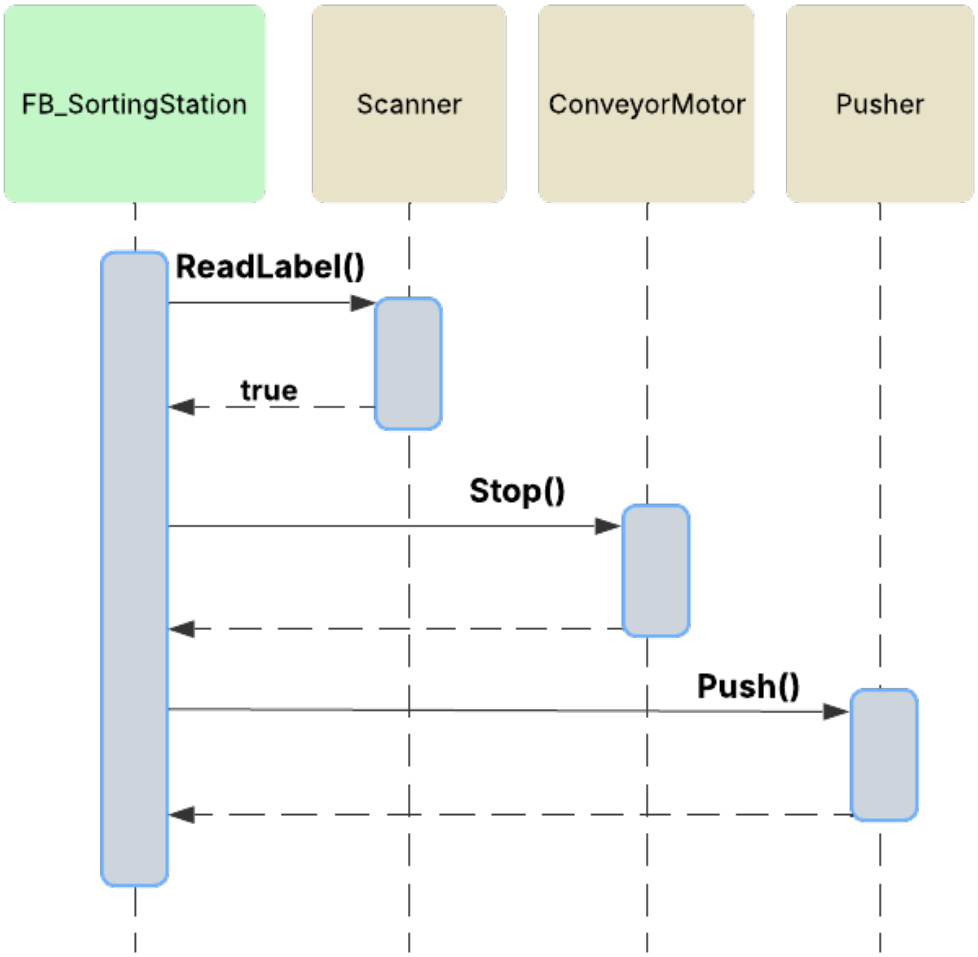


Figure 2.4: Sequence diagram example.

### *2.5.4 Applicability to IEC 61131-3 and PLC-development*

UML was designed for object-oriented software systems and the third edition of IEC 61131-3 introduced, as previously mentioned, constructs to support this: Functions Blocks with methods, interfaces, inheritance and encapsulation. This maps directly to the UML core notation. A Function block corresponds to a UML class, extends corresponds to inheritance, while implements corresponds to interface realization. Integrating IEC 61131-3 with UML enables the use of multiple diagram types to capture different aspects of the system, creating a more complete and comprehensive model (Thramboulidis and Frey, 2011).

## 2.6 Core Architectural Components

Industrial automation control systems share a set of common structural and functional responsibilities that recur across machine types, application domains and PLC platforms. These include managing the operational state of the machine, handling abnormal conditions, processing physical signals and defining how the machine stops and recovers. How these responsibilities are organized and how they interact with each other has a direct impact on the readability, maintainability and scalability of the resulting codebase (John and Tiegelkamp, 2010).

### *2.6.1 Equipment Modules*

In the ISA-88 batch control standard, groupings of equipment that can carry out a finite number of specific minor processing activities are referred to as Equipment Modules (ISA, 2010). While ISA-88 defines this concept specifically to batch manufacturing, the underlying principle, that a distinct physical subsystem should correspond to a distinct, self-contained software unit, is broadly applicable across automation domains.

In this thesis, the term Equipment Module (EQM) is used in this broader sense to refer to any modular unit responsible for encapsulating the control logic of one section of the machine.

### *2.6.2 Machine State-Coordination*

Industrial machines rarely operate in a single, undifferentiated mode. A typical machine progresses through a sequence of distinct operational phases, each with its own associated behavior and transition condition. Explicitly representing this structure is a fundamental requirement of well-designed automation software (John and Tiegelkamp, 2010).

A state machine is a formal model that describes a system in terms of a finite set of discrete states, the events or conditions that trigger transitions between them, and the actions associated with each state or transition (Object Management Group, 2017). As introduced in section 2.5.2, state machine diagrams provide a platform-agnostic means of specifying this behavior before implementation, mapping directly into CASE-based ST logic in the PLC environment. Code example 2.8 shows a simplified state machine used to control a sorting

station. An enum, previously described in 2.2.4.1, named 'SortingState' is used to define the set of valid states.

```
VAR
    SortingState : (IDLE, RUNNING, FAULT);
END_VAR
METHOD Execute
    CASE SortingState OF //Per PLC-cycle, only current
       ↪ states logic executed
        IDLE:
            IF StartCommand THEN
                SortingState := RUNNING;
            END_IF;
        RUNNING:
            IF StopCommand THEN
                Stop();
                SortingState := IDLE;
            ELSIF Fault THEN
                SortingState := FAULT;
            ELSE
                Run();
            END_IF;
        FAULT:
            IF Acknowledged THEN
                Reset();
                SortingState := IDLE;
            END_IF;
    END_CASE;
END_METHOD
```

Code Example 2.8 – State machine of a sorting station.

In a machine composed of multiple independent subsystems, each subsystem may maintain its own internal state. Still, the machine as a whole must also progress through a shared operational sequence in a coordinated manner. A subsystem that begins its startup sequence before the safety system has confirmed a safe state, or one that continues to run after a stop request, represents a failure of coordination. State coordination is therefore the mechanism by which the overall machine state is managed centrally, ensuring that each component operates within the boundaries defined by the current machine phase.

In the PLC context, state machines serve a practical function beyond structural clarity. Because the PLC executes cyclically as described in section 2.4, the state machine provides

a mechanism for distributing behavior across scan cycles in a controlled and predictable way. Each scan, the current state determines which logic executes, and transition conditions are evaluated to determine whether the state should change on the following scan (Antonsen, 2020).

### *2.6.3 Alarm & Event Handling*

IEC 62682 is a standard that defines an alarm as an audible or visible indication of a condition requiring operator attention or action (SEK Svensk Elstandard, 2023). An event is a broader term referring to any notable occurrence within the system, including state changes and informational notifications that do not necessarily require a response. The distinction matters because not every condition warrants the same machine reaction, and a functional alarm system must be capable of expressing this variation consistently.

A possible approach for representing alarms is using a struct, previously described in 2.2.4.2, composed of appropriate variables. Code example 2.9 shows the struct 'AlarmEvent' as well as how the previously mentioned FB_SortingStation could check alarm conditions and populate the instance of AlarmEvent called 'MotorAlarm'.

```
TYPE AlarmEvent :
STRUCT
    ID    : INT;
    Active    : BOOL;
    Message   : STRING;   //Message to machine operator
    Severity : (WARNING, ERROR, CRITICAL); //Indicates how the
       ↪  machine should proceed
   //...Further AlarmEvent-attributes
END_STRUCT
END_TYPE

//Method inside a FB_SortingStation, runs every PLC-cycle
METHOD CheckAlarms
    IF Motor.Fault THEN
        MotorAlarm.Active := TRUE;
        MotorAlarm.Message  := 'Motor fault on conveyor 1';
        MotorAlarm.Severity := WARNING;
    END_IF;
    //....Further logic to raise sorting station alarms
END_METHOD
```

Code Example 2.9 – AlarmEvent struct applied in FB_SortingStation.

### *2.6.4 Recipe Handling*

In industrial automation, recipes represent the process-specific parameters, which vary between products produced with a machine. The primary standard used for recipes is ISA-88, which separates recipes into a hierarchical structure: general, site, master, and control recipe. The general recipe describes the manufacturing process at a conceptual level, while the site recipe adapts it to the capabilities of a specific site. The master recipe then ties it to a specific machine, making it ready for execution, while the control recipe is the active instance created when a production starts (ISA, 2010).

The parameters for a recipe could be the speed of a motor, the temperature of a heating element, or an ingredient amount. Code example 2.10 shows a possible implementation of a recipe for a sorting station and how the recipe parameters conceptually get applied.

```
TYPE SortingRecipe :
STRUCT
    MotorSpeed : REAL;   // belt speed
    Lane1BarcodePrefix : STRING; //Ties product with specific
       ↪ barcode to be routed to the correct lane
    Lane2BarcodePrefix : STRING;
    //...Further recipe parameters for more sorting logic
END_STRUCT
END_TYPE
//Method inside FB_SortingStation
METHOD ApplyRecipeParameters
    Motor.SetSpeed(SortingRecipe.MotorSpeed);
    //...apply remaining parameters
END_METHOD
```

Code Example 2.10 – Sorting station recipe.

### *2.6.5 Safety Integration*

Functional safety in industrial automation refers to the portion of the overall safety of a system that depends on the correct operation of safety-related control functions. The IEC 62061 and ISO 13849 standards define frameworks for the design and assessment of safety-

related control systems, classifying safety functions according to the level of risk reduction they must provide (IEC, 2021).

In the PLC context, safety integration concerns how safety-related signals and functions are incorporated into a broader control architecture. Depending on the safety integrity level required, this may involve dedicated safety-related hardware and a separate safety PLC running certified safety logic, with the main PLC receiving status information from the safety system rather than processing safety signals directly (IEC, 2021). A simplified example of this is shown in code example 2.11, where the motor is instantly shut off if a received safety signal indicates that a fault occurred.

```
// Running on the safety PLC
METHOD EvaluateSafety
    SafetyOK := NOT EmergencyStop
              AND NOT GuardDoorOpen
              AND NOT LightCurtainTriggered;
END_METHOD
// Running on the main PLC
METHOD MotorSafetyCheck
    IF NOT SafetyOK THEN
        MotorRunning  := FALSE; //Shuts down motor if received
           ↪  safety signal indicates faulty operation
        Fault:= TRUE;
    END_IF
END_METHOD
```

Code Example 2.11 – Safety signals.

### *2.6.6 Stop & Recovery Handling*

Industrial machines must be capable of stopping in a controlled manner in response to a range of conditions, from operator-initiated production stops to emergency conditions triggered by alarms previously mentioned in 2.6.3. The nature of the stop required depends on the condition that triggered it and the potential consequences of different stopping strategies (IEC, 2016).

### *2.6.7 Input/Output Handling*

As described in section 2.4, the PLC does not read physical inputs continuously during program execution. Instead, all input values are captured into a process image at the beginning of each scan cycle and held stable throughout the execution of the program. Output values are written to the process image during execution and transferred to the physical output modules at the end of the scan. In IEC 61131-3, variables are bound to process image addresses using the 'AT' keyword followed by a direct address (IEC, 2025), as shown in Code Example 2.12. The address consists of a location prefix ('%I' for input, '%Q' for output), a size prefix ('X' for bit), and a byte.bit number. Meaning MotorFaultSignal AT %IX0.0 reads from input byte 0, bit 0 of the process image each cycle, and MotorRunSignal AT %QX0.0 writes to output byte 0, bit 0 at the end of it.

```
1 VAR
2  MotorRunSignal      AT %QX0.0 : BOOL;//signal to start motor
3  MotorFaultSignal    AT %IX0.0 : BOOL;//fault signal from motor
4 END_VAR
```

Code Example 2.12 – Input and output signals for a motor.

# 3. METHOD

This chapter describes the research methodology and process applied throughout the thesis. Section 3.1 presents the overarching research approach and its three guiding cycles. Section 3.2 outlines the work phases. Section 3.3 describes the development process. Sections 3.5-3.7 cover the data collection methods: the literature review and the semi-structured interviews. Section 3.8 addresses source criticism.

## 3.1 Research Approach

The thesis proposal from Etteplan showed a design-oriented approach, where the primary goal was the development of a reusable template, accompanied by the appropriate documentation, for use by their engineers, based on Etteplan's needs and the surrounding literature. This naturally called for a research methodology centered around design and development. Therefore, Design Science Research (DSR) was chosen as the overarching research approach, as it is specifically suited for research where the result is intended to solve real-world problems (Hevner et al., 2004).

Unlike empirical research, which seeks to describe or explain a phenomenon, DSR evaluates how well the produced result, called 'artifact', solves the problem it was designed to solve. Hevner et al. (2004) describe DSR through its three core cycles:

- The Relevance Cycle – aims to ground the research in a real-world need.
- The Rigor Cycle – anchors and connects the research to existing knowledge and theory in its field.
- The Design Cycle – describes the iterative process of development and evaluation of the artifact itself.

These cycles were applied in this thesis, extended into the sub-phases outlined in section 3.2 and further discussed in 3.3.

## 3.2 Work Phases

The three cycles of DSR were extended to the five work phases, seen in Figure 3.1 and discussed in this section. The work phases were not planned to be conducted linearly, as the thesis had to allow for an iterative workflow. Except for phases one and five, each phase's prerequisites and expected outcomes were therefore not explicitly stated beforehand.

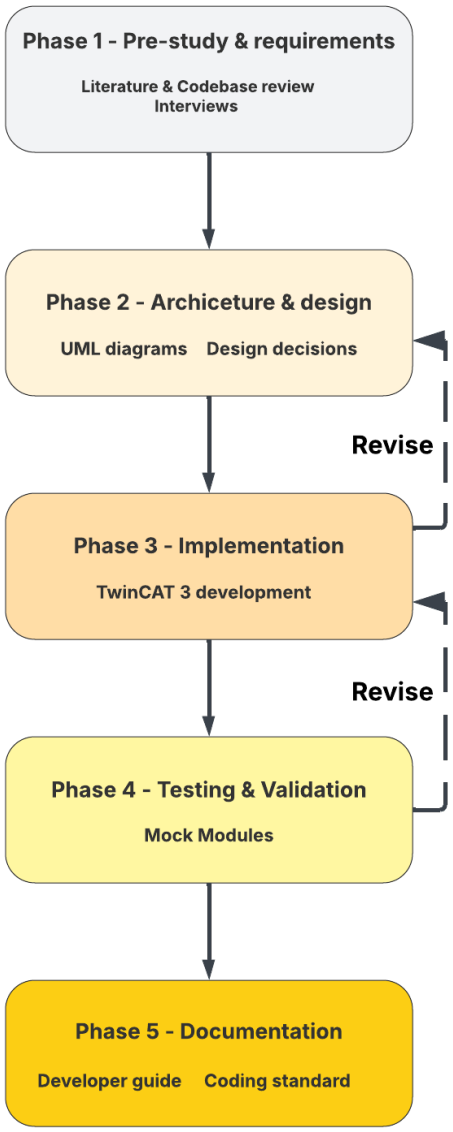


Figure 3.1: Thesis work phases.

***Phase 1 – Pre-study & Requirements***

Etteplan provided a document describing their proposal for the thesis and what they expected to be achieved. This was the starting point for phase one and gave a rough idea of what more research needed to be done.

During this phase, the literature review was conducted, and then, with the preliminary results of the literature review, a topic guide for the interviews was created. The interviews were held for several weeks. Meanwhile, phases two and three started.

***Phase 2 – Architecture & Design***

During phase two, the results of the literature review were used to create preliminary design concepts of the template. This was done with UML diagrams describing the hierarchy of class inheritance, sequence diagrams and state diagrams to visualize and map the needed functionality of each part.

As previously mentioned, this was iterated continuously back from phase three in order to apply new insights gained from the active implementation. This caused the original design decisions in phase two to be heavily redesigned when phase five started.

***Phase 3 – Implementation***

In this phase, the core components of the template were implemented in TwinCAT 3, while continuously backtracking to previous phases to iteratively improve the design.

***Phase 4 – Testing & Validation***

Phases four and five were conducted somewhat in parallel to each other, allowing for continuous testing of the template using simple mock modules, simulating the behavior of real equipment. This was the primary reason for backtracking to previous phases when a tested design did not perform as expected.

***Phase 5 – Documentation***

During the fifth phase, the final implemented design was represented using UML diagrams and the accompanying documentation was created, necessary for the use of the template.

## 3.3 Development Process

The development process in this thesis did not follow a specific formal methodology. Instead, the work was conducted in a flexible and iterative manner, primarily guided by the principles of DSR and its three cycles, as previously described.

Hevner (2007) describes the relevance cycle as the establishment of a connection between the research to be conducted and the environment in which it is located. This ensures that the research addresses a meaningful and practical problem and sets the theoretical foundation for producing an artifact capable of being applied and tested in that environment. In this thesis, the relevance cycle is primarily applied in phase one, but with a slight extension into phase two, where models of the artifact had to be grounded and molded iteratively in accordance with Etteplan's expectations.

In the same manner as the relevance cycle, the rigor cycle was also primarily adapted into phases one and two. The rigor cycle is the natural complement of the relevance cycle, linking research with the existing knowledge base in the research field, including theories, frameworks, models, and previous studies (Hevner, 2007). Since the relevance and rigor cycles were conducted simultaneously during phase one and part of phase two, they naturally evolved and formed each other. For example, the topics covered by the interviews were grounded in the literature reviews, and the respondents' answers, in some cases, promoted an extension and continuation of the search for relevant literature. Hevner (2007) discusses this phenomenon when conducting research aligned with DSR. He contends that descriptive theories produced in the rigor cycle must be allowed to be merged and grounded with the result of the relevance cycle. Otherwise, the artifact designs are not research contributions, but merely routine designs based upon well-known processes.

The design cycle is the core iterative process in which the artifact is designed, developed, and evaluated. In this design process, the two inputs are derived from the two previous cycles, requirements from the relevance cycle. The rigor cycle provides the design and evaluation method, and this iteratively produces self-improving artifacts (Hevner, 2007). In this thesis, the design cycle is extended to phases two, three, and four. As described previously and

visible in Figure 3.1, a developed artifact from phases two and three is evaluated in phase four, and then iteratively revised back into phase two, until satisfactory.

## 3.4 Communication

The thesis was carried out by two students working in close collaboration throughout the project. Work was divided equally with regular alignment in order to ensure consistency across the shared codebase and report. To facilitate this, a shared Git repository served as the primary tool for coordination of code contributions, and Google Docs was used for other file handling and report writing. Most of the time spent on the thesis was done at Etteplan's office with both students present, but when not present in the office, a digital communication app was used to coordinate work for the project.

At Etteplan, a supervisor was assigned to the thesis, and meetings with the supervisor were done weekly and when deemed necessary by either party. This allowed for opportunities to present progress, collect feedback, and discuss encountered difficulties, but also to ensure the work remained aligned with the company's needs. The feedback received from the supervisor at Etteplan during the meetings was documented and, where relevant, incorporated into the subsequent iterations of the template and its accompanying documentation.

## 3.5 Literature Review

The literature review was conducted primarily during the pre-study phase and this served two purposes. The first was to establish how modern PLC software is developed by identifying existing standards, design principles and practices relevant to the thesis. The second purpose was to provide a theoretical foundation for the design choices made throughout the thesis project, ensuring that the design of the PLC framework was a combination of established knowledge from the literature and the answers the respondents gave in interviews.

Literature was primarily sourced using the Lund University search engine 'Finn', as well as Google Scholar. Search terms included 'IEC 61131-3', 'PLC object-oriented programming', 'modular PLC design', as well as terms related to the core components outlined in chapter 2.6. The selection of chosen sources prioritized peer-reviewed journal articles, and these usually

cited standards relating to specific design or architectural patterns for PLC systems used in manufacturing. These standards, such as IEC 61131-3 or ISA88, were used as authoritative primary sources given their role as the defining framework of PLC programming practices in industry. Furthermore, textbooks and peer-reviewed articles were used to establish the research methodology, interview technique and data analysis.

The literature review was conducted as a semi-structured review (Snyder, 2019), directed by the research problems as well as the evolving design decisions during design and development. This means that though the primary search result for literature was helpful as a starting point, further research was necessary during the development phase. These sources were selected based on their direct relevance to the topics in the project, such as state machine design, alarm handling, or recipe handling.

## 3.6 Interviews

As a complement to the literature review, semi-structured interviews were conducted with engineers at Etteplan. The interview's purpose was not to perform a broad qualitative study, but instead to capture practitioner knowledge, industrial experience and the expectations of the result of this thesis. This was deemed not readily available in academic literature, specifically the pain points and workflows engineers experience in their daily work. The result of the interviews partly created justifications for the design decisions made throughout this thesis.

Semi-structured interviews were chosen as the format, which allowed for a prepared topic guide while still leaving room for conversation to develop naturally based on the respondent's experience (Kvale and Brinkmann, 2009). This format was considered more appropriate than fully structured interviews, which can result in a rigid and narrow interview, which might not allow for the nuance of the individual respondents' experiences.

The interviews were conducted with five engineers at Etteplan, with different roles within the organization. Including two junior automation engineers and three senior automation engineers. This variation in roles was deliberately chosen to capture the experience and expectations of what a standardized template should prioritize. Each interview lasted ap-

proximately 60 minutes and was either conducted in person or digitally, depending on the respondent's availability. During the interviews, notes were taken as well as audio recordings, with the consent of the respondent.

The material was then transcribed and analyzed using the thematic analysis approach by Braun and Clarke (2006). A thematic analysis consists of steps, starting with the raw, transcribed data. From that initial similarities were identified. Amongst the similarities shared by respondents, potential themes were reviewed, categorized and named. These themes are further outlined in section 4.1.

## 3.7 Testing & Validation Methods

The testing and validation of the validity, reliability and usability of the produced iterations of the template was primarily done with mock-EQMs. These were used to test whether a specific or broader part of the template acted as expected when given data as if by a real machine. The mock-EQMs were designed, mostly based on the behavior of the code exhibited in the PLC projects provided by Etteplan. They were also designed, in part, based on the literature of outlined standards, if applicable. This was complemented by the continuous feedback given by the Etteplan supervisor, whose knowledge and experience of the behavior of physical machines influenced the design of the template.

Hevner et al. (2004) discusses this approach for evaluation within the research methodology, DSR. They mean that an artifact can be evaluated in several metrics, such as functionality, completeness, and accuracy. This can be done with several methodologies, but the selected methodology must be appropriate for the evaluation of a specific artifact. The testing and validation for the template were based on what Hevner et al. (2004) categorized as 'Experimental' and 'Descriptive', using simulated data for experimental validation. The descriptive testing was then based on the result of the literature review, interviews and the supervisor's expertise.

This approach to test and validate the development and result of the template has several flaws. Because the template was designed based on the provided code from Etteplan and the interviews, the transferability and generalizability of the template and documentation are

primarily limited to the context of Etteplan's engineers. However, the architectural design of the framework also had the context of the literature review, and was designed in accordance with the latest edition of IEC 61131-3. In theory, this would make the template transferable and valid between PLC platforms, but this was not tested. Furthermore, the template was never tested with physical hardware.

## 3.8 Source Criticism

Sources were selected based on their relevance to the research domain. Standards such as IEC 61131-3 and the Object Management Group (2017) specification serve as primary, authoritative sources. Academic publications were assessed by peer-review status and citation count. Textbooks by recognized authors in software engineering and PLC programming were also used, cross-referenced with recent literature where applicable.

IEC 61131-3 is the primary technical authority of the thesis. Its 4th edition (IEC, 2025) was published recently, and industry adoption remains limited at the time of writing. References to its OOP constructs may therefore not yet reflect established practice.

SEK Svensk Elstandard (2023), the Swedish adoption of IEC 62682, is used as the primary source on alarm management and safety. ISA-88 is the primary source for recipe and equipment module concepts. The Oracle Java SE specification (Oracle Corporation, 2026) is treated as authoritative for Java language constraints referenced by analogy.

Academic literature, including Hevner et al. (2004); Hevner (2007),Braun and Clarke (2006), and Thramboulidis and Frey (2011), consists of peer-reviewed journal articles and conference proceedings. The literature specifically addressing OOP in the context of IEC 61131-3 is sparse, which limits the academic foundation for some design decisions. Textbooks by Antonsen (2020), John and Tiegelkamp (2010), Freeman and Pryce (2009), Gamma et al. (1994), and Kurose and Ross (2012) served as key references. Antonsen's work is closely aligned with the thesis scope but predates the 4th edition and does not cover its OOP extensions.

A general limitation is the modest volume of literature directly addressing the intersection of OOP and industrial PLC development at the architectural level. Some design decisions

therefore rest on established software engineering principles adapted to the PLC context, rather than directly validated PLC-specific research.

## 3.9 Use of AI in Thesis

Throughout this thesis, AI tools have been used. Anthropic's Claude has been used to aid in programming, helping with the usage of proper syntax and adhering to existing code standards. Claude was also used to improve grammar and correct spelling while maintaining consistency throughout the thesis. NotebookLM was used to cross-reference the literature and navigate the resources. The authors of this thesis take full ownership of the material produced and assert that AI was used only as a support and that all text and code were written by themselves.

# 4. ANALYSIS

This chapter presents the analysis of the gathered information. Section 4.1 presents the thematic analysis of the interviews. Sections 4.2 and 4.3 present the framework analysis, grounded in the codebase review, literature, and interview findings.

## 4.1 Interviews

Thematic analysis resulted in seven themes, listed in Table 4. Each theme, coded T1 to T7, is analyzed in sections 4.1.1 to 4.1.7. Respondents are coded R1 to R5.

The themes informed design decisions throughout sections 4.2 and 4.3, and were used to answer the research questions and confirm the problem formulation. Respondents generally gave similar answers, which may reflect shared professional context rather than independent convergence.This is a limitation discussed further in section 3.7.

Table 4: Interview themes.

| Code | Theme name | Consensus |
|---|---|---|
| T1 | The legibility imperative | All five respondents |
| T2 | Architecture as communication | All five respondents |
| T3 | The legacy burden | R3, R4, R5 |
| T4 | Reuse as a strategic priority | All five respondents |
| T5 | The standardization paradox | All five respondents |
| T6 | The expert-novice knowledge gap | All five respondents |
| T7 | Testing as a secondary concern | All five respondents |

### *4.1.1 The Legibility Imperative*

The 'legibility imperative' theme is essentially the combined overarching super-theme of all subsequent themes. However, specific nuances discussed by respondents R3 and R5, which did not fit subsequent themes. Therefore, this theme captures the respondents' recurring conviction that PLC code must be written and maintained to be understood by people other than the developer.

Within this, two distinct, but related expressions of legible code were discussed by the respondents: intra-team legibility and cross-domain legibility. Here, the former includes the original

developers, future engineering colleagues, junior developers, and the latter non-programmer service technicians.

All respondents discussed several coding practices that affected the intra-team legibility of PLC code. The specifics of such practices were generally seen as unimportant, but they were defined and universally applied. Some of these coding practices were the foundation for subsequent themes in the thematic analysis.

Respondents R3 and R5 extended this into cross-domain legibility where the applied coding practices had to be altered to include code legibility for non-engineers. This was described by R5 as the main reason why the use of LD persists, to facilitate code legibility for technicians, while ST would have been technically superior and perhaps preferred by the engineers. Respondent R3 attributed intra-domain legibility in part to the slow progression of the utilization of OOP principles in PLC-programming, and if it were to be used, it had to be limited to code that service technicians would not encounter.

### *4.1.2 Architecture as Communication*

'Architecture as communication' captures the idea that the way a PLC program is organized (throughout several dimensions) functions as a shared language that allows engineers to orient themselves quickly in unfamiliar code. Encompassing why the structural consistency within a codebase is valuable, especially if the implementation details differ.

The respondents had varied opinions on the details of such an architecture and its implementation. Respondents R1, R2 and R3 focused on a sub-system folder architecture. Where everything logically related to, for example, recipes, is kept. R1 elaborated on this structure and extended it into where code is executed, describing the Main POU as the crest of a waterfall, where all other subsystems are called. Creating a starting point for where code is meant to be read from.

Two respondents, R4 and R5, emphasized some pitfalls of a highly abstracted architectural structure. Such codebases could be architecturally elegant, but the original purpose of communicating logic through the structure is lost in deeply nested structures. R5 describes the initial disorientation of entering such an architecture for the first time as structurally opaque,

and communicating nothing.

### *4.1.3 The Legacy Burden*

'The legacy burden' captures the reality that PLC programs in industrial settings are rarely designed from scratch. Instead, they are inherited, extended and maintained across years or decades, often by engineers with no shared standard and no access to the original authors' intent.

Such an experience was mostly shared by the more senior developers, respondents R3, R4 and R5. Respondent R5 offered the most extreme case, describing a production codebase approximately 20 years old, where code written before any organizational standard coexists alongside newly standardized code. R5 partly attributed this to engineers building on each other's previous codebases without a unifying architecture. Respondent R4 addressed this problem in relation to new implementations of OOP having to be retrofitted on top of a non-OOP codebase, or large sections of code having to be rewritten to facilitate OOP principles.

### *4.1.4 Reuse as a Strategic Priority*

'Reuse as a strategic priority' captures the shared conviction among all five respondents that writing the same code more than once is wasteful. All respondents discussed mechanisms for structured reuse of code that would be helpful in their daily work, in different ways.

All five respondents identified device-level reuse, through FB's for standard device types such as inverters, servos, and valves, as the highest priority reuse mechanism. Respondent R3 illustrated this principle in brief as "build the FB once, use what you need from it later". With a different perspective, R2 mostly discussed structural reuse through templates and a defined code architecture. This was emphasized by R3 and R5, such as a reuse mechanism with recognizability as its primary goal. Respondent R2 acknowledged that the dominant mechanism, device-level, has a core limitation of transmitting bad patterns alongside good ones. Reuse of functional code, applied in a new environment, sometimes is inappropriate because of the structural or version differences between the environments.

### *4.1.5 The Standardization Paradox*

'The standardization paradox' captures the tension, expressed by all five respondents, between the desire for greater standardization and the structural forces that consistently undermine it.

At the PLC-vendor level, respondent R5 described IEC 61131-3 compliance among vendors as practically unreliable, with implementation differences of the IEC 61131-3 standard varying greatly between them. Meaning shared concepts between vendors cannot be assumed to behave identically between platforms. R4 extended this, describing genuine cross-platform standardization as years of work followed by continuous maintenance. R2 also identified lack of such maintenance would risk 'template decay', noting that unmaintained templates become sources of bad patterns rather than good ones.

At the organizational level, R1 described standards that exist but are inconsistently applied, while offering the clearest resolution: the specific content of standardization matters less than its universal application.

### *4.1.6 The Expert-Novice Knowledge Gap*

'The expert-novice knowledge gap' captures the recurring concern about what happens when expert knowledge fails to transfer, whether to a junior engineer, a service technician, or a future engineer inheriting code with no handover.

Two complementary knowledge-transfer mechanisms were identified. Structural knowledge, concerning where and how code is placed hierarchically. This was emphasized by respondents R3 and R5. R3 described the primary value of a well-structured template as the elimination of structural uncertainty. R5 extended this to shorten the onboarding of junior engineers by having a structural starting point, embedded with senior-level knowledge. The other identified knowledge-transfer mechanism was pattern knowledge. Concerning how code logic is conveyed, this was emphasized by R2 and R4. R2 identified the most useful tools for a junior engineer as: self-describing code and, if necessary, descriptive inline comments. R4 recommended external documentation with code examples, especially for bigger projects.

### *4.1.7 Testing as a Secondary Concern*

'Testing as a secondary concern' captures the gap, present across all five respondents' interviews, between the stated value of systematic testing and the actual testing practices in use. Usually characterized by hardware dependence, proprietary tool development and manual execution testing.

No respondent described systematic, automated testing as an organizational standard. R1 described manually removing hardware dependencies to run programs locally. R3 described the most structured approach: a three-tier system combining virtual PLC software, a hardware simulator, and physical machine testing before each release. R3 also identified unclear initial specifications as the single greatest source of testing problems. R5 framed the issue economically, contrasting the high cost of machine rental against cheaper simulation alternatives. This suggests that the barrier to proper testing is monetary rather than a lack of motivation from engineers.

## 4.2 General Architecture

An analysis of source code from previous Etteplan projects, complemented by insights gathered through the interviews, revealed a range of differing architectural approaches across implementations. These variations reflect the natural diversity that arises when projects are developed independently by different engineers over time. Each represents a functioning solution to the problem at hand. To fully leverage the capabilities outlined in IEC 61131-3 and establish a consistent foundation for future development, a set of architectural guidelines was defined for the framework. These guidelines are primarily informed by the strengths that OOP principles bring to PLC development. They are intended to improve efficiency, scalability and maintainability for new projects going forward.

A central outcome of the design process was the establishment of guidelines for when and how the various programming constructs defined in IEC 61131-3 should be applied within the framework. These decisions were informed by the requirements gathered during the pre-study, the literature review, the interviews conducted with Etteplan engineers and the analysis of existing project source code.

### *4.2.1 Function Blocks as the Structural Foundation*

The distinction between PRGs and FBs follows directly from the roles defined in IEC 61131-3. While the IEC standard defines PRGs as the top-level execution unit as described in section 2.2.1, their use as a container for logic introduces a significant constraint; a PRG can only exist as a single instance within a project. This means that any logic placed directly in a PRG cannot be reused or repurposed with different implementations. For a framework whose primary goals are reusability and scalability, this constraint is incompatible with the intended design.

FBs, on the other hand, support multiple instantiations that encapsulate their own state, can implement interfaces and be extended through inheritance, as described in section 2.3. In the framework, even the top-level coordinator is an FB, with the MAIN program reduced to a single call containing no logic of its own. This means that the entire machine program is designed to be instantiable, testable and substitutable. In practice, an engineer can instantiate the machine FB in a test harness, swap out individual components for mock implementations through their interface contracts, and validate behavior without deploying to hardware. This is illustrated in Figure 4.1.

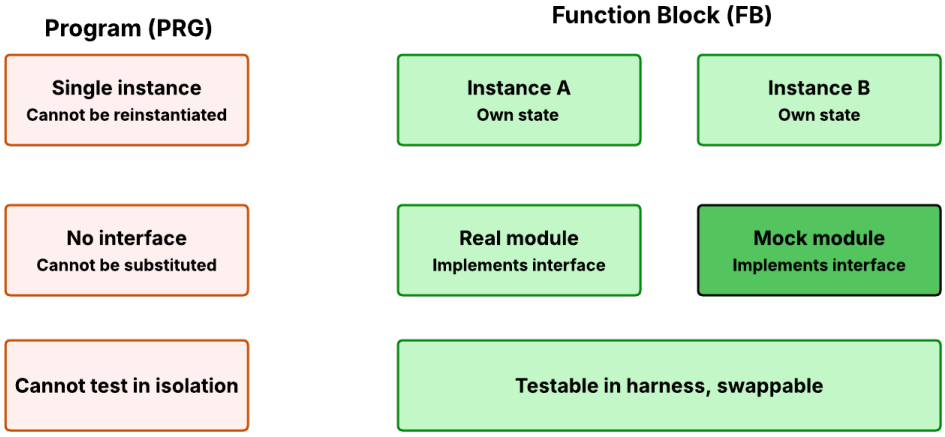


Figure 4.1: Program vs Function Block.

Methods decompose the internal behavior of a Function Block into purposeful, self-explanatory units. Rather than placing all logic directly in the FB body, the framework uses methods to keep the body readable as a sequence of named operations, while the details of each opera-

tion are encapsulated within its respective method. This is consistent with the encapsulation principle described in section 2.2.2. A servicing engineer encountering the codebase for the first time can understand the overall behavior of a component from its method call sequence alone, without needing to read through the full logic inline.

This is the pattern the framework strives towards, and it is most valuable where the FB body would otherwise become long or contain logic spanning several distinct responsibilities. For small, single-purpose FBs where the body logic is concise and self-evident, placing logic directly in the body remains acceptable and may in fact be clearer than introducing methods for the sake of consistency alone.

### *4.2.2 Inheritance: Enforcing Contracts Through Abstract Methods*

Inheritance is applied in the framework when a set of Function Blocks share a common structural contract and some shared infrastructure, while differing in their specific behavior. The alternative was duplicating shared logic across multiple independent FBs, but this was rejected because it reintroduces exactly the kind of redundant reimplementation identified in the pre-study as a cause of poor maintainability. When the same logic is copied across every module in a project, any change to that logic must be applied in every copy, with no guarantee of consistency.

Shared infrastructure is placed in the base FB, while derived FBs implement abstract methods for the parts that vary. This ensures that common behavior is defined once and inherited reliably, with the compiler enforcing what each derived FB must provide rather than leaving conformance to convention. This arrangement reflects the template-method pattern defined by Gamma et al. (1994), through which consistency becomes a structural property of the architecture rather than a responsibility delegated to individual engineers. In practice, adding a new Equipment Module to a project is guided directly by the abstract method declarations, requiring no study of prior implementations to determine what the framework expects.

The inheritance hierarchy in the framework is deliberately kept shallow. Chains deeper than two levels introduce implicit dependencies: a derived FB inherits not only from its immediate base, but from every ancestor above it. A change to any base class can therefore

propagate unexpectedly through the hierarchy, and understanding a derived class requires tracing through every ancestor in turn (Martin, 2014). This directly contradicts the goal of reducing navigation time for engineers unfamiliar with the codebase. Where deeper hierarchies might otherwise arise, composition is preferred as an alternative, as described in section 2.3.5 and illustrated in Figure 4.2.

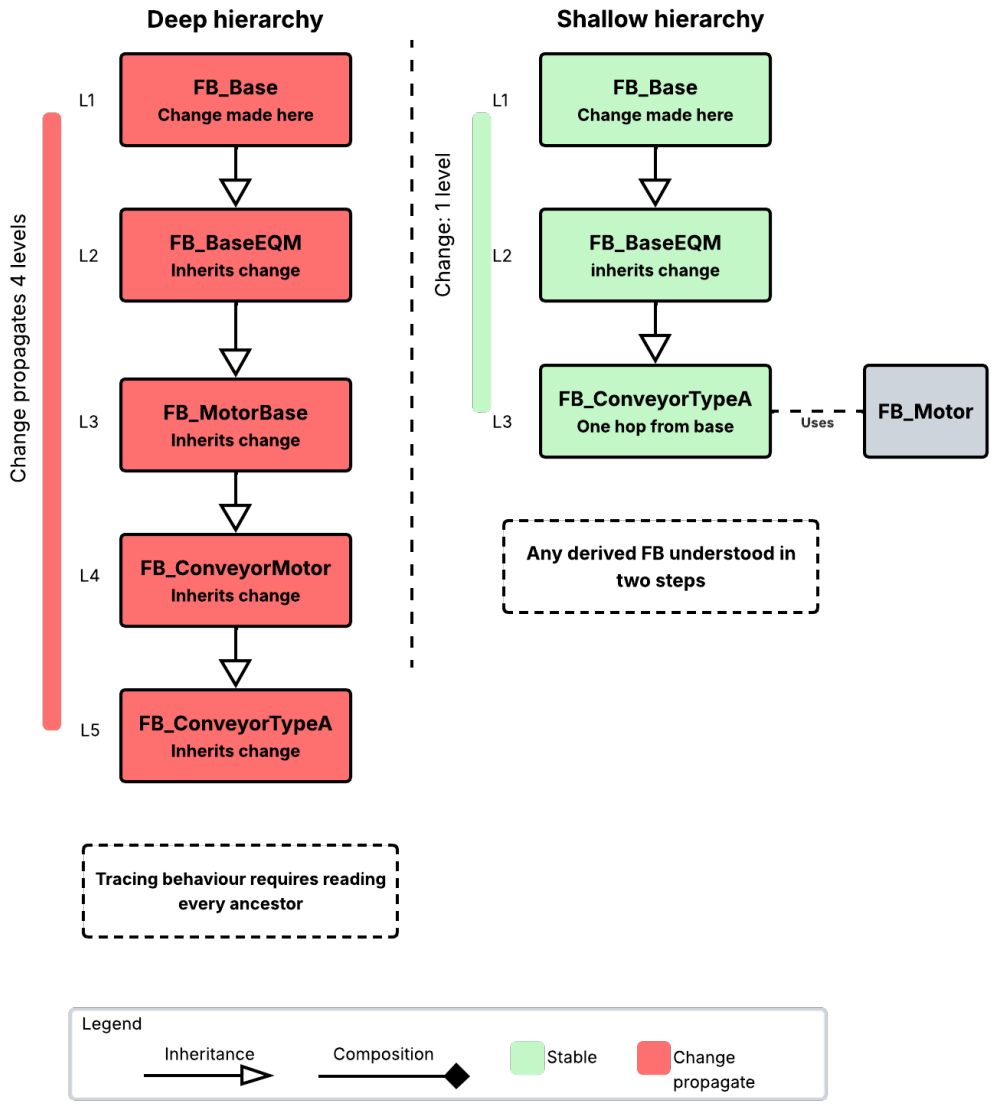


Figure 4.2: Deep vs Shallow inheritance.

### *4.2.3 Interfaces: Enabling Substitutability*

While inheritance expresses what a component is, an interface expresses what it can do. Based on the functionality established in section 2.3.3, interfaces are used in the framework

wherever one component needs to communicate with another without being coupled to a specific implementation.

This is motivated by the enabling of substitutability between real and simulated implementations. A recurring finding in the pre-study was that PLC programs in existing projects had no mechanism for simulation or isolated testing, because modules were directly instantiated and called as concrete types. If the calling code holds an interface reference rather than a concrete instance, any FB that satisfies that interface contract can be substituted without modifying the calling code. A hardware-dependent module can therefore be replaced by a mock implementation for testing and restored for deployment without modifying any surrounding logic. This is illustrated in Figure 4.3.

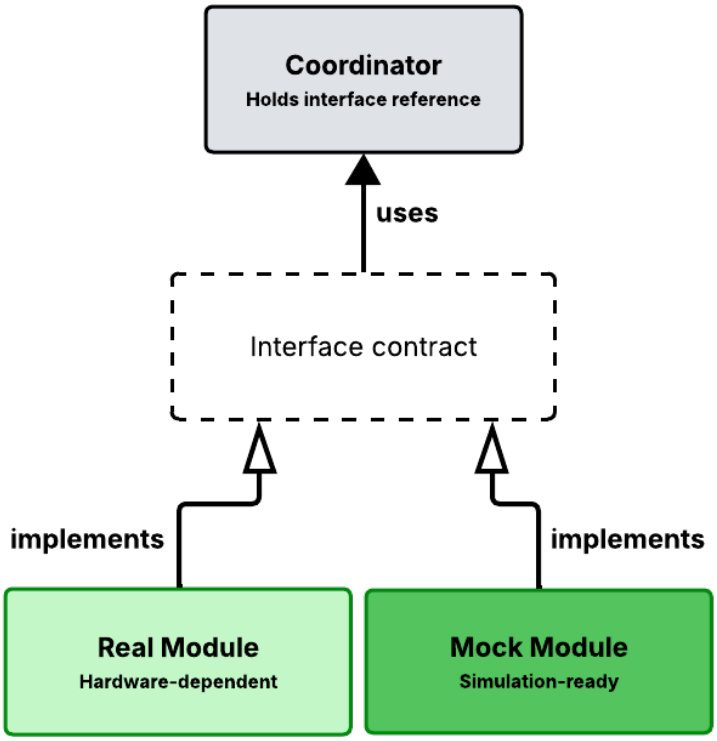


Figure 4.3: Interface Substitutability Illustration.

All interface methods in the framework are defined without properties. This decision was made to ensure compatibility across all target platforms, as property accessors are not uniformly supported across TIA Portal and Studio 5000 in the same way as in TwinCAT 3. Replacing properties with equivalent GET methods preserves the intended behavior while

keeping the interface contract portable.

### *4.2.4 Exposing Outputs to the Rest of the Program*

How an FB exposes its results to the surrounding program has direct consequences for readability and coupling. Two approaches were considered: exposing outputs as declared variables wired at the call site, or returning results through method outputs. The framework adopts the former. Wiring outputs at the call site keeps data flow visible and explicit, without requiring the caller to invoke additional methods to retrieve results. An engineer reading an FB call can immediately identify its inputs and outputs, without navigating into the FB's internal implementation.

The other approach was using GET methods, which are methods with the sole purpose of outputting an internal variable from an FB. These are reserved for cases where a derived FB needs to expose a value that is computed from a combination of internal state variables and cannot naturally be represented as a static output. This distinction keeps the more common case simple and readable, while still accommodating derived or conditional values where needed.

### *4.2.5 Ownership and Instantiation*

The template establishes that a component should own and instantiate whatever it is directly responsible for coordinating. FB instances are declared at the level where they are called and managed, not created externally and passed in. This keeps the composition of the system visible in a single location and removes the ambiguity that arises when instances are created at one level and consumed at another. The pre-study code analysis identified the opposite pattern as a contributor to unclear program structure.

Dependency injection through interface references is reserved for components that may need to be substituted between deployment contexts. In these cases, holding an interface reference rather than a concrete instance allows the implementation to be exchanged without modifying the consuming component (IEC, 2025). The two mechanisms are complementary: interfaces define the contract, and injection provides the means of supplying an alternative implementation at runtime.

### *4.2.6 State Machines*

Representing the operational phases of a component as an explicit state machine, rather than through boolean flags and nested conditional logic, was a deliberate choice grounded in both the literature and the pre-study findings. As discussed in section 2.6.2, state machines make the discrete operational phases of a component explicit, along with the conditions that govern each transition.

Expressing component behavior as a state machine ensures that the operational state of any module is unambiguous at any point during execution. Transition conditions are co-located with the state definitions themselves, reducing the cognitive load required to understand module behavior and making faults easier to trace during commissioning and servicing.

The framework does not prescribe a single state machine pattern for all components. The machine-level coordinator and the individual Equipment Modules operate at different levels of abstraction with different transition semantics. The appropriate state machine structure for each is left to the engineer using the framework.

### *4.2.7 Structs and Enumerations*

Global variables, as introduced in section 2.2.4.3, were explicitly avoided as a means of passing information between components. The codebase review identified extensive use of global variables as a primary contributor to unclear dependencies, where the origin and ownership of a value were difficult to determine from the code alone. The framework instead uses structs to bundle related data and enumerations to constrain variables that can only take a finite set of named values.

Enumerations are used wherever a variable is restricted to a defined set of named values, such as machine states, stop types, command types, or event severities. As described in section 2.2.4.2, the compiler enforces the valid range of an enumeration-typed variable, transferring responsibility for correctness from the programmer to the type system. In a framework maintained by different engineers over time, this is particularly valuable: invalid assignments are caught at build time rather than at runtime.

Structs group related fields into named, typed units passed as a single value between compo-

nents. This reduces the number of individual variables managed at each call site and allows fields to be added without modifying the call signatures of the FBs that use the struct. That extensibility was considered essential for keeping the framework evolvable without cascading changes across the codebase.

## 4.3 Machine Core Components

Given the core components proposed by Etteplan (section 1.1) and their industrial relevance (section 2.6), an overarching hierarchical view of the components and their interconnections is presented in Figure 4.4. Each subsection presents the analysis of the data gathered from the literature and codebase reviews and interviews. Through a combination of these analyzes, the most important design decisions are discussed and presented in their corresponding sections.

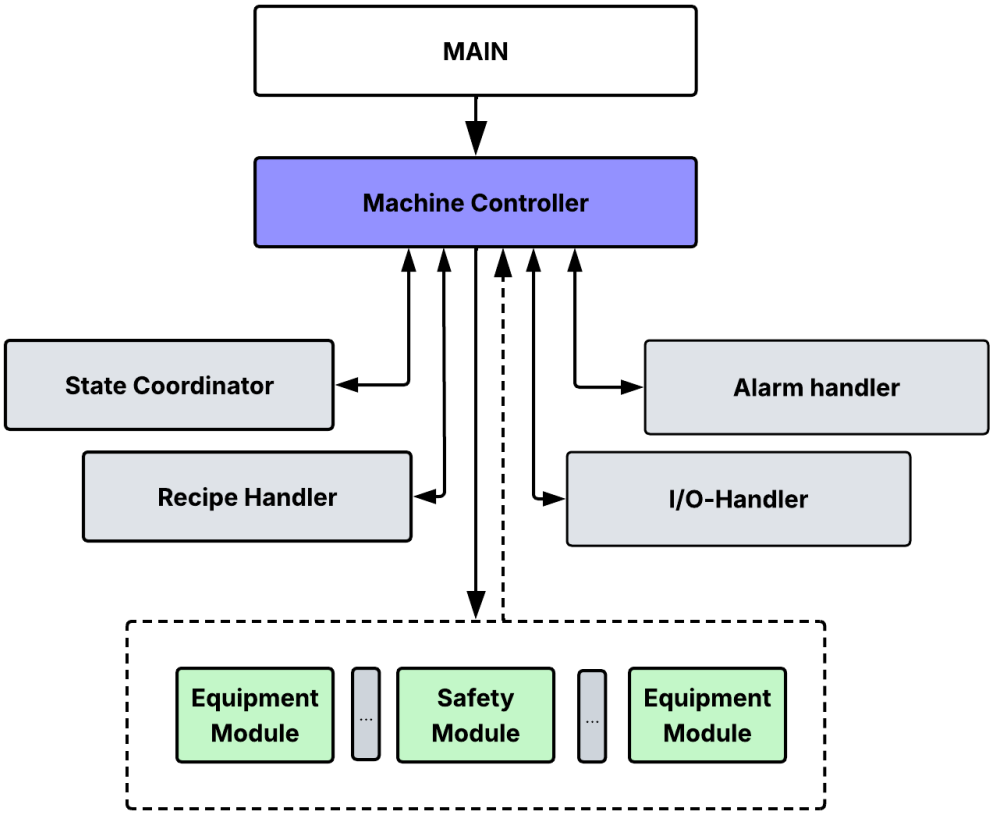


Figure 4.4: Overarching view of the template.

### *4.3.1 Equipment Module Base*

A recurring finding in the pre-study interviews was that similar device and subsystem logic is reimplemented independently across projects, with no shared foundation enforcing consistency between implementations. The Equipment Module Base addresses this by establishing a common structural contract that all machine subsystems must satisfy, regardless of the specific logic they implement.

The design decisions needed for the base EQM were primarily:

- What is the common functionality amongst EQMs
- How shared infrastructure should be separated from module-specific logic
- How EQMs should communicate to the rest of the system.

#### *4.3.1.1 Equipment Module Commonalities*

Based on the pre-study result, commonalities between EQMs were identified. Therefore, the template should implement the minimal necessary structure for the identified common subsystems seen in Table 5. While still allowing the user to modify each subsystem.

Table 5: EQM commonalities.

| Structural subsystem | Description |
|---|---|
| State machine | Module advances its internal state |
| Alarm instances | Populates, triggers and passes own alarms |
| Recipe status | Passes recipe status to and from the handler |
| HMI update | Passes data for the operator display |

#### *4.3.1.2 Defining the Contract Through Abstract Methods*

The decision to implement abstract methods in the base FB reflects a deliberate choice about where responsibility should lie. An alternative would have been to provide a default implementation for these methods in the base, allowing the derived FBs to override them only when needed. This approach was rejected because it would permit a module to be

instantiated without providing any module-specific behavior, making it possible to add a module to a project that compiles and runs but does nothing meaningful. By declaring these methods as abstract, the framework transfers the enforcement of completeness from documentation and convention to the compiler.

This pattern also has implications for onboarding. An engineer implementing a new module for a project does not need to study prior implementation to understand what is required; the abstract declarations enumerate the obligations explicitly. This directly addresses the knowledge transfer problem identified in the pre-study, where the absence of structural guidance was cited as a primary cause of long onboarding times and inconsistent implementation across teams.

#### *4.3.1.3 Separating Initialization from Cyclic Execution*

IEC 61131-3 does not permit dynamic object creation during runtime (IEC, 2025). All instances and their configurations must be established before cyclic execution begins, or, where a dedicated startup task is unavailable, at the initial scan. This initialization separation also improves the readability of the base's body. An engineer reading the body sees a short, fixed sequence of method calls representing the module's per-scan responsibilities, as seen in code example 4.1: state machine, alarm instances, recipe status and HMI update. The one-time configuration logic is contained in a distinct initialization block, keeping it separate from cyclic execution. This reflects the broader framework convention that the body of a function block communicates the overall structure of the component's behavior, and that implementation detail is contained within named methods, consistent with the encapsulation principle outlined in chapter 2.3.1.

```
FUNCTION_BLOCK FB_BaseEQM
VAR
END_VAR
IF NOT initialized THEN
     Init();
END_IF
 Statemachine();
 AlarmInstances();
 RecipeStatus();
 HMIUpdate();
END_FUNCTION_BLOCK
```

Code Example 4.1 – BaseEQM body.

### *4.3.2 State Coordinator*

The machine state coordinator manages the overall operational lifecycle of the machine and governs when and how transitions between states occur.

Three design questions shaped this component:

- How to separate coordination mechanics from the specifics of any given machine's phase sequence.
- How to safely aggregate readiness across modules before allowing a transition.
- How to propagate stop requests consistently to all modules.

#### *4.3.2.1 Separating Coordination Mechanics from Phase Logic*

A first design option was to implement a single State Coordinator that encodes a specific phase sequence, used as-is across all projects. The limitation of this approach is that different machine types have fundamentally different operational lifecycles. A packaging machine with a cleaning and heating phase has different state requirements from an assembly robot cell, and a coordinator hardcoded to one sequence cannot serve the other without modification. Any modification to the coordinator to accommodate a new machine type risks introducing errors in the shared mechanics of readiness checking and stopping propagation.

The template addresses this by separating what the coordinator does from the specific states it transitions through. The concrete phase sequence is delegated to derived FBs through an abstract method, while all coordination mechanics remain as concrete methods in the base. Modules, therefore, share identical coordination infrastructure while defining entirely different phase sequences. Adding a new machine type requires only a new implementation of the phase method. This is consistent with the template-method design pattern described earlier in section 4.2.2, and directly supports the scalability requirement identified across all five interview respondents.

#### *4.3.2.2 Module Readiness*

The state coordinator must determine whether all EQMs are collectively ready before permitting a phase transition. Interview consensus pointed clearly toward a coordination mechanism that evaluates readiness per subsystem. This is a core scalability property: transition logic does not change when a new EQM is added.

Two complementary readiness mechanisms are appropriate. The first reads a Boolean readiness flag exposed as a direct output by each EQM. The second queries readiness through the downward-facing interface contract, requiring each EQM to implement a readiness method. Each scan, the coordinator advances the machine state only when all registered modules satisfy the readiness condition. Either mechanism is valid; the template implements one. This is illustrated in Figure 4.5.

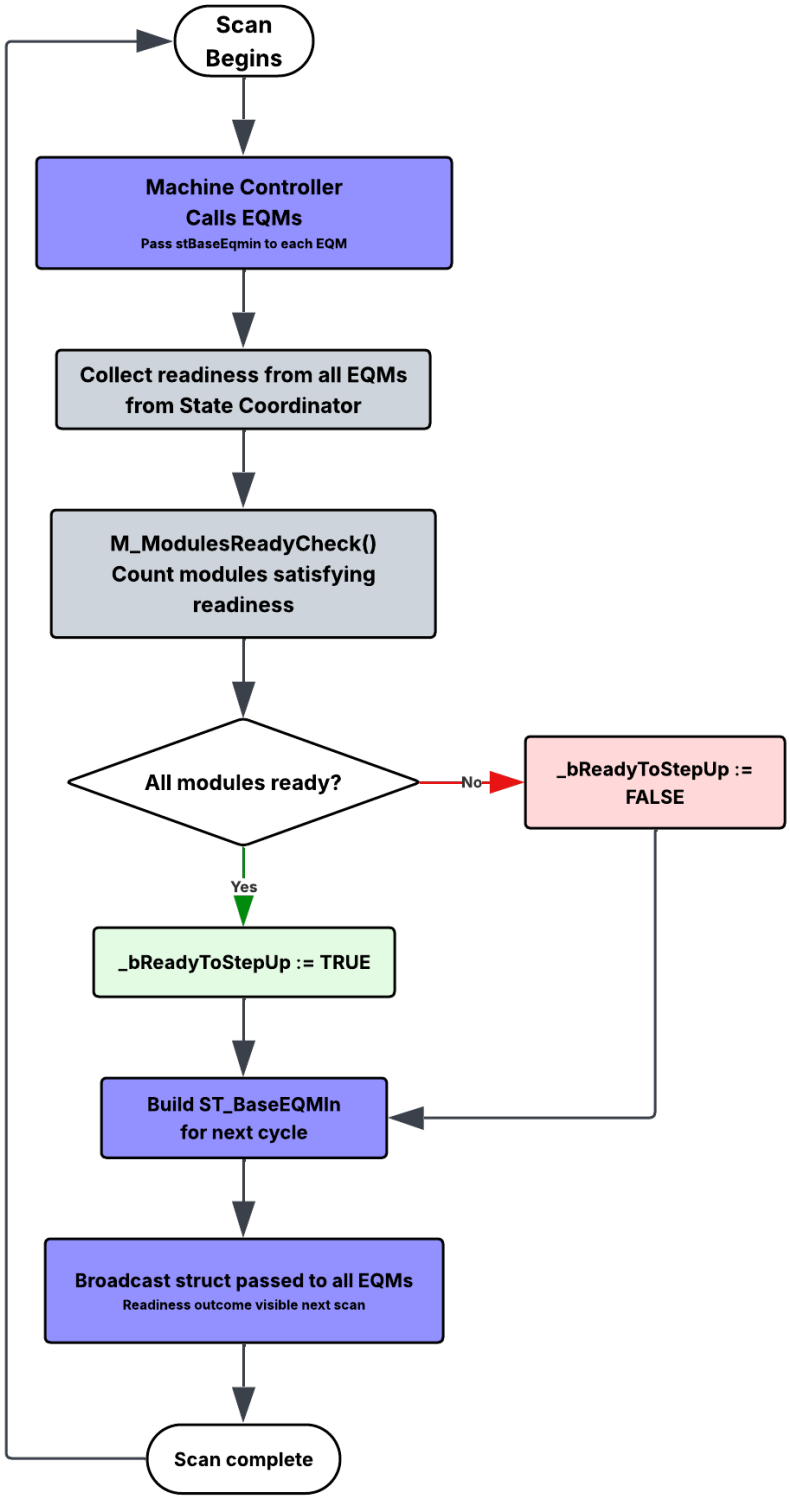


Figure 4.5: Module readiness flowchart.

#### *4.3.2.3 Top-level Program*

The main program serves as the task entry point required by IEC 61131-3. It contains no logic of its own. The entire machine is encapsulated within a single FB, instantiated here and called once per scan. This follows directly from the structural principles in section 4.2: a Machine Controller FB is fully instantiable and testable in isolation without hardware.

Because the Machine Controller is a function block, it can be instantiated inside a test harness alongside simulated modules. EQMs implementing the downward-facing interface can be replaced by mock implementations at any point without modifying the Machine Controller or State Coordinator. This makes hardware-independent substitution a structural property of the framework, directly addressing the pre-study finding that tight hardware coupling was among the most significant barriers to logic verification.

The Machine Controller communicates with EQMs through a broadcast struct, shown in Code Example 4.2. All EQMs receive the same machine-level information each cycle.

```
TYPE ST_BaseEqmIn : // Common inputs to BaseEQM modules
STRUCT
 eMachineState  : E_MachineStates; // From StateHandler
 nMachineState  : USINT; // numeric alias
 bAckAlarms     : BOOL; // From AlarmHandler
 eStopRequest   : E_StopTypes;//Stop request
 bSafetyOK      : BOOL; // FALSE triggers eAborting in EQM
 //... Further broadcast logic
END_STRUCT
END_TYPE
```

Code Example 4.2 – Inputs to EQMs from State Coordinator.

### *4.3.3 Alarm Handler*

In industrial automation, the control system must detect and communicate abnormal machine behavior to the operator through alarms. Depending on severity, an alarm may trigger a machine state transition to allow safe identification and resolution of the cause.

The alarm handler provides consistent orchestration of alarm data flow, decoupling alarm

detection in EQMs from HMI presentation, while supplying the State Coordinator with a reliable view of the current alarm state. Figure 4.6 illustrates this flow.

Four design decisions govern alarm handling in the template:

- Which responsibilities belong to EQMs, and which are delegated to the alarm handler?
- How are alarms represented?
- How are alarms raised?
- How are alarms communicated outward?

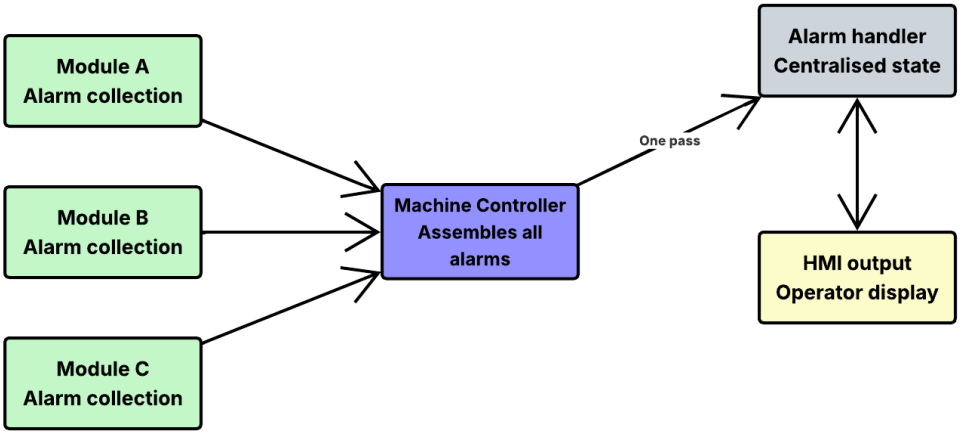


Figure 4.6: Alarm data flow.

### *4.3.3.1 Centralized or Distributed Handling*

During the codebase review, several approaches were identified for managing alarms in machines. These were based upon two opposite approaches: a distributed model where EQMs fully manage their own alarms, or a centralized model where a separate handler manages raised alarms. Usually, it was a combination of the two, depending on the machine scope or function.

The distributed approach can be appropriate in smaller systems, where every alarm an EQM can raise is predetermined at design time and integrated into the full system, including the HMI. For a framework intended to scale across machine types of varying complexity, however,

the centralized model is more appropriate. The centralized model also aligns with the IEC 62682 standard, which establishes alarm management as a system-level concern, rather than a component-level one, as outlined in section 2.6.3.

#### *4.3.3.2 Alarm Structure and Lifecycle*

Each alarm is represented as a structured object within an EQM, containing the attributes necessary to describe a distinct alarm condition. The template defines the minimum required attributes; specific trigger conditions are user-defined. The alarm struct is the standardized representation used throughout the template: populated by EQMs, processed by the alarm handler, and presented via HMI. This aligns with IEC 62682 requirements that each alarm carry sufficient information to identify the condition, state, priority, and required operator response (section 2.6.2).

Alarms are declared explicitly before execution. This is partly a consequence of the IEC 61131-3 execution model, which does not permit dynamic object creation at runtime (IEC, 2025). It is also a deliberate design principle: explicit declaration ensures all alarm conditions are configured before execution, supporting the alarm rationalization process recommended in IEC 62682 (SEK Svensk Elstandard, 2023).

```
TYPE ST_Event : // Event information to send to alarm handler
STRUCT
    AlarmActive: BOOL;
    AlarmUnAcked: BOOL;
    StopRequest: E_StopTypes;
    StateRequest: E_MachineStates;
    Severity: E_EventSeverity; // Event severity.
    Text: STRING;// Alarm text
    ID: DINT;// Alarm ID
    Module: STRING;// alarm belongs to
    TimeStamp: DATE_AND_TIME;// Time when  triggered
END_STRUCT
END_TYPE
```

Code Example 4.3 – Alarm struct.

#### *4.3.3.3 Raising of Alarms*

With the centralized design of the template's alarm handler, the design of how EQMs communicates alarms to it had to be defined. The codebase review revealed several approaches on how this can be designed, and all seem somewhat equally appropriate, but also hold disadvantages.

One option is that the EQM holds a reference to the alarm handler via an injected interface. The EQM calls a method directly on the alarm handler reference. EQMs could either pass one alarm object as an argument when an alarm state changes or pass a collection of alarm objects directly to the handler. This approach is consistent with the dependency inversion principle; the EQM depends on an abstraction rather than a concrete handler type, keeping the coupling indirect and extendable (Martin, 2008).

Another option for the template should be that each EQM exposes a collection of its alarm states as an output variable. The machine controller should collect these, assembling them into a unified collection from all EQM's and pass them further to the alarm handler. This design allows the unified collection to be received by the alarm handler in one coordinated pass. The tradeoff of this approach is that alarm state objects pass through several intermediate data structures before reaching the alarm handler, introducing more moving parts than the injection model.

#### *4.3.3.4 Communicating Alarms*

In each program cycle, the alarm handler should receive updated alarm objects from each EQM, and from this, it has two responsibilities. Maintaining a system-wide view of active alarms, communicated to the State Coordinator, as well as forwarding alarm information to the HMI.

The alarm handler should expose a collection of alarm data to the State Coordinator, limited to only the necessary information the State Coordinator requires for state transition logic. This limiting of data to the State Coordinator is based on the Interface Segregation Principle (Martin, 2008), restricting the State Coordinator to consume the alarm handlers' exposed

collection rather than full internal access to all alarms. How the State Coordinator should use these collections to govern state transitions is primarily user-defined, but the framework defines minimum requirements, previously discussed in 4.3.2.

For the HMI, the alarm handler should forward active alarm entries containing their specific data fields to be presented to the operator. Within the template, this should be done by exposing alarm data at a well-defined output; how that data is read and interpreted by the HMI is user-defined. This is a deliberate design decision as communication between PLC and HMI varies between platforms and projects. Therefore, the template's separation between the alarm handler and HMI prevents the embedding of such logic and should contribute to its reusability. This separation is also aligned with IEC 62682 identification of HMI as a distinct boundary within an alarm system architecture, separate from alarm detection and alarm state management.

### *4.3.4 Recipe Handler*

The recipe handler orchestrates the transition between stored and active recipes, distributing and validating recipe data for EQMs. Three design decisions governed its design:

- How should recipes be represented?
- How should the recipe handler communicate with EQMs?
- Where should recipe validation be performed?

#### *4.3.4.1 Recipe Format and Storage*

How recipe data is stored and loaded is user-defined. Recipes may be hardcoded, loaded from external files, or fetched from a database via the HMI. The template must accommodate all of these without prescribing a specific approach. Furthermore the template should be compatible with recipe parameters of different data types, to allow the application to different machine types.

The design draws on the ISA-88 standard's recipe model. Full ISA-88 recipe separation was deemed out of scope, but the template introduces a distinction between a stored, inactive

recipe struct and an active, loaded recipe, analogous to ISA-88's master and control recipes. A recipe first exists as a stored object and transitions to active once loaded and validated into an EQM. This is described further in sections 4.3.4.2 and 4.3.4.3.

Recipes are defined by the user within EQMs, following the same approach used for alarms in section 4.3.3.2. The template provides a base recipe struct with common attributes, shown in Code Example 4.4.

```
TYPE ST_BaseRecipe :
STRUCT
    nRecipeID       : INT;     // Unique identifier
    sRecipeName     : STRING;  // For HMI display
    sVersion        : STRING;  // Version string
    bIsValid        : BOOL;    // Set by validation logic
    sValidationMsg  : STRING;  //Message for HMI
    sCreatedBy      : STRING;  //ID of creator
    sModifiedDate   : STRING;  // Last modification
END_STRUCT
END_TYPE
```

Code Example 4.4 – Base Recipe struct.

### *4.3.4.2 Recipe Handler Communication*

A push-based distribution model was selected over a pull-based model. In a pull model, EQMs fetch recipe data independently, bypassing intermediate validation. The push model gives the recipe handler control over the distribution sequence, enabling validation before delivery.

EQMs must register with the recipe handler at initialization, giving the handler a registry of which modules it is responsible for. This follows the mediator pattern (Gamma et al., 1994): the recipe handler decouples those that load recipe data from those that consume it, so neither must reference the other directly.

Communication between the recipe handler and EQMs includes a handshake protocol. EQMs expose flags indicating whether a recipe has been received, validated, or faulted. This is necessary because IEC 61131-3 provides no constructs for asynchronous callbacks (IEC, 2025).

The handshake is modelled on acknowledgement protocols in computer networking, where the receiver informs the sender of the status of transmitted data (Kurose and Ross, 2012).

#### *4.3.4.3 Recipe Validation*

Validation of recipes is generally unique to the recipe data, which should be user-implemented and therefore out of the scope for the recipe handler in the template. However, the template must be able to orchestrate the validation sequence between the recipe handler and the EQMs. The template should also propose which type of validation is appropriate for the recipe handler and which should be done by EQMs. This aligns with the Single Responsibility Principle (Martin, 2008), such that the validation logic within the user-implemented recipe handler should not need to change as a recipe data structure changes. Therefore, validation responsibility should be distributed to where the relevant data is held. Furthermore, the validation logic of the recipe handler should be limited to structural failure of the data. For example, missing fields or wrong data types. The validation logic of the EQMs should be limited to the specifics of that EQM, for example, a value that is within the data type's range, but not valid based on the limitations of what the physical hardware within the EQM can handle. While this is not fully compliant with ISA-88, the design principle of validation against the equipment capabilities before execution (ISA, 2010) is derived from it.

The recipe handler's conceptual functionality and its interaction with other core components are visualized in Figure 4.7. Showing descriptive transition criteria between its four main states, as well as one fault state.

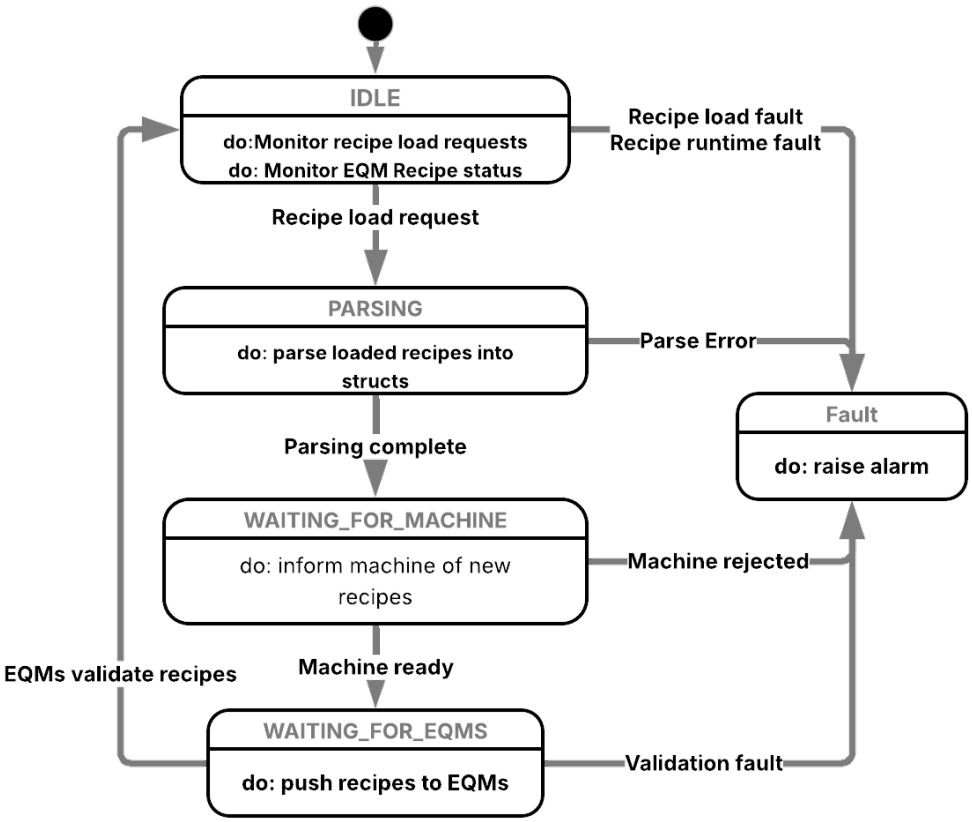


Figure 4.7: Recipe handler state machine.

### *4.3.5 Safety Handler and Stop Recovery Handling*

In industrial automation, a machine operates across a range of conditions, such as normal production, startup, shutdown, and fault states. During fault conditions, the control system must bring the machine to a known-safe state, meaning actuators de-energized, moving parts halted, and process sequences terminated in a defined order. How quickly and in what sequence this must happen depends on the nature of the fault. A pre-study finding shared across all respondents, and consistent with the requirements established in IEC 62682 (SEK Svensk Elstandard, 2023), is that this stopping behavior must be defined and traceable at the architectural level, not distributed as ad-hoc conditional overrides across individual EQMs.

The appropriate stopping response depends on the type of machine the framework is deployed on. ISA-88 distinguishes between process types whose modules operate independently in sequence and those whose modules cooperate to maintain a continuous process (ISA, 2010).

Two archetypes illustrate the range of requirements.

In a line machine, material moves sequentially through discrete stations. When one station faults, stop propagation is directional: upstream modules must halt to prevent accumulation, while downstream modules can continue until their buffers empty. Respondents of the pre-study interview with experience in sequential machine environments noted that forcing a simultaneous halt across an entire line risks discarding in-process material and extending recovery time beyond what the fault condition requires.

In a process machine, all modules cooperate to maintain a continuous physical or chemical process, such as a controlled reaction requiring stable temperature, pressure, and flow. A fault in any one module threatens the whole process, and all modules must stop together in a coordinated sequence. Partial stopping risks an uncontrolled condition that no single module can independently recover from.

These archetypes produce fundamentally different stopping requirements from the same framework. This is why stopping behavior cannot be prescribed at the framework level. Each EQM defines its own stopping sequence, since what constitutes a safe stop depends on the physical devices it controls and its role in the broader machine. IEC 62682 requires that such stopping behavior be defined and traceable at the architectural level rather than distributed as ad-hoc logic across individual modules (SEK Svensk Elstandard, 2023).

The design questions for this component of the framework, therefore, concern:

- How the safety status should be communicated to the rest of the system,
- How different fault conditions should map to different stopping responses
- How individual EQMs should define and execute their own stopping sequences while remaining coordinated at the machine level.

#### *4.3.5.1 Safety Coordination*

As established in section 2.6.5, safety integration in a PLC system concerns how safety-related signals and functions are incorporated into the broader architecture. The safety

system and the standard control logic operate as distinct layers, with the standard PLC receiving status information from the safety system rather than processing safety signals directly. The interface between these two layers is therefore a boundary that the architecture must define explicitly: the standard control system must be able to read the status of the safety system and respond by transitioning to an appropriate safe state, while the safety system itself remains independent and continues to function regardless of the state of the standard control logic.

The first design decision, therefore, concerned where in the architecture safety monitoring should be located, and how safety status should cross this boundary into the rest of the system. Binding the Machine Controller directly to safety hardware signals would couple the Machine Controller to the specific safety hardware configuration of a particular machine, making it impossible to reuse the Machine Controller across projects with different safety architectures. Implementing safety monitoring as an EQM that extends the same abstract base as all other EQMs eliminates this coupling. This makes the Safety Handler act as an EQM, which participates in the readiness gate and raises alarms through the same alarm infrastructure. Furthermore, this Safety Module is registered with the Machine Controller in the same way as all other EQMs.

The Safety Module additionally exposes a public method that the Machine Controller calls each scan when assembling the broadcast struct, making the safety status available to every EQM simultaneously through the struct's dedicated field. This field is the only channel through which safety status influences EQM behavior. An EQM that reads a false value in this field can initiate an immediate abort sequence, but the abort sequence itself is defined within the EQM's own state machine, preserving the autonomy of each EQM over its own stopping behavior. A proposed sequence of how a safety signal propagates is visualized in Figure 4.8.

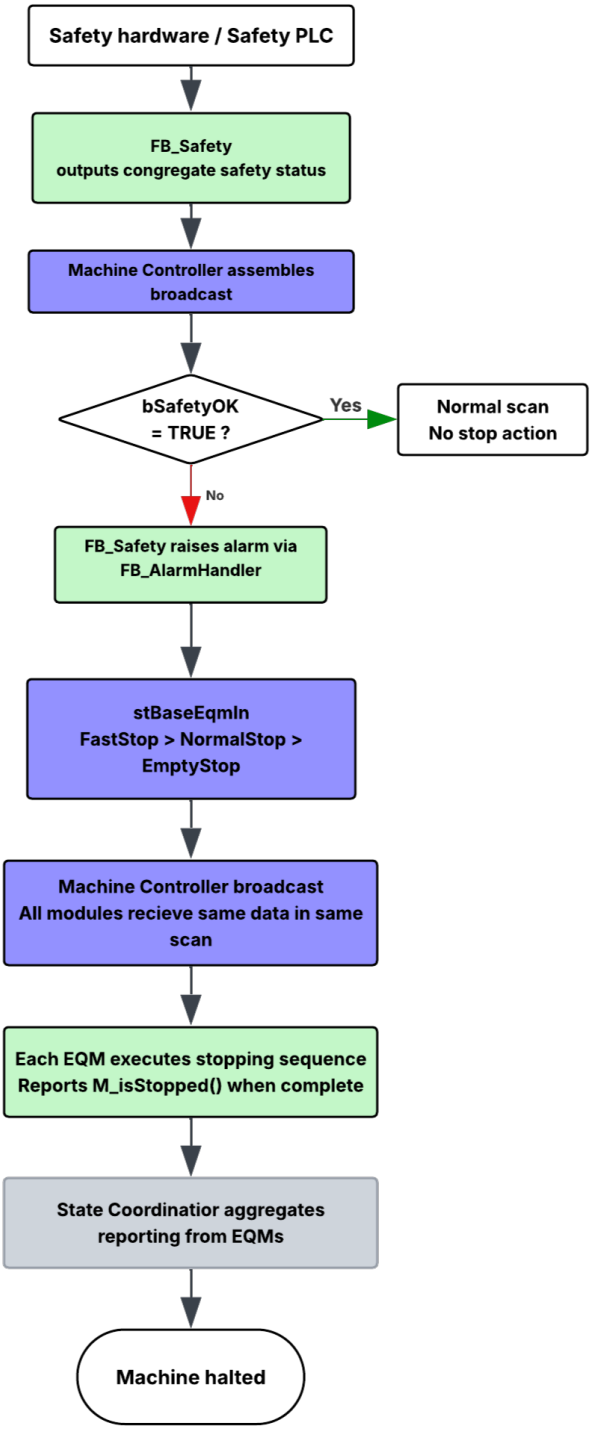


Figure 4.8: Lifecycle of a Safety signal.

Treating safety as a structural participant in the coordination protocol rather than as a

special case override has two practical consequences. First, adding, removing, or modifying safety channels requires only changes within the Safety Module itself. No other EQM, State Coordinator, or Machine Controller needs to be modified, as they interact with the Safety Module only through the readiness interface and the broadcast struct field. Second, the Safety Module can be replaced by a mock implementation for testing purposes using the same substitution mechanism that applies to any other EQM, allowing the Machine Controller and EQMs to be tested in scenarios where the safety system reports either a normal or a faulted state without requiring physical safety hardware.

#### *4.3.5.2 Stop Types and Fault Severity*

Informed by the findings during the literature review and interviews, the template was developed with an enumeration composed of three defined types ordered by urgency, based on the standard IEC 60204-1. Empty stop, which permits the current production cycle to complete before the machine halts; a normal stop, which stops the machine when subsystems have reached a safe intermediate state; and a fast stop, which commands all EQMs to halt as quickly as their physical constraints permit (like in the case of an emergency stop) (IEC, 2016). Each alarm instance is configured with a stop type at initialization. When the alarm becomes active, this is reported to the alarm handler and, after priority resolution by the Machine Controller, broadcast to all EQMs via the broadcast struct. This approach was derived from the results of the interviews as the validated industrial pattern on the machines they work on.

Using an enumeration rather than a boolean flag or a numeric literal for the stop type, per code example 4.5, provides the same compiler-enforced constraint described for other enumerations in section 2.3.6: only values within the defined set can be assigned, making it impossible to specify an unrecognized stop type through a coding error (John and Tiegelkamp, 2010). The ordered structure of the enumeration further allows the priority resolution logic in the Machine Controller to select the highest-priority active stop type through a simple conditional chain, without requiring a lookup table or numeric comparison that would be opaque to a reader unfamiliar with the priority ordering. Martin (2014) identifies implicit

numeric semantics of this kind, where the meaning of a value depends on knowledge not visible at the point of use, as a common source of maintenance errors in codebases that grow beyond their original scope.

```
TYPE E_StopTypes :
(
    FastStop,
    NormalStop,
    EmptyStop
    //.. further user-defined stoptypes
)
END_TYPE
```

Code Example 4.5 – Enumeration of stop types.

#### *4.3.5.3 Stopping Behavior as an Abstract Contract*

The framework requires each EQM to define its own stopping and aborting sequences through abstract methods declared in the base. This reflects the single responsibility principle described by Martin (2014): an EQM stopping behavior is a consequence of the physical devices it controls, and only the EQM itself has the information necessary to define what a safe stop means in its context. A conveyor motor may require a controlled deceleration before power is removed; a pneumatic actuator may need to reach a defined position before the air supply is cut; a temperature-controlled process may need to hold its state until a safe threshold is reached before shutdown. None of these specifics can be anticipated by the template.

Declaring these methods as abstract rather than providing default implementations has the same enforcement motivation described for the equipment module base in section 4.3.1.2: a module that does not define its stopping behavior cannot be compiled, which transfers responsibility for completeness from documentation and code review to the compiler. Cruz Salazar and Vogel-Heuser (2020) identify the enforcement of structural contracts through language constructs rather than through convention as one of the principal safety-relevant arguments for OOP in the IEC 61131-3 context, since it eliminates a class of omission errors that are otherwise detectable only at runtime or during commissioning. In a safety-critical context,

the cost of discovering a missing stopping sequence at commissioning is substantially higher than the cost of addressing a compiler error during development.

### *4.3.6 Input/Output Handler*

The codebase review revealed frequent use of physical I/O addresses directly being used as references in logic components. Meaning, reading sensor inputs and writing actuator outputs within the same code that processes such values. This approach creates a tight coupling between internal logic and the hardware currently in use. Any change to the physical wiring, I/O address assignment, or PLC platform could require extensive changes throughout the internal logic itself. It can also make testing or simulation of internal logic difficult without the physical hardware present.

The framework requires separation of physical I/O addresses and how they are applied for internal processing logic. Furthermore, it should give the user the necessary functionality for the communication of user-defined internal variables to and from EQMs. Two design decisions were needed:

- Where should the responsibility of I/O-separation be located?
- How are user-defined variables coupled to physical I/O-addresses?

#### *4.3.6.1 Ownership of I/O Mapping*

Two approaches to I/O separation were identified during the codebase review. Either each EQMs performs such separation locally for its own concerned variables, or a dedicated mapping layer above the EQMs handles I/O separation.

The local separation approach for each EQM was omitted quite early during the design. This design fundamentally does not align with the Single Responsibility Principle (Martin, 2008); EQMs should only be responsible for behavior, not the separation of physical wiring or hardware components. It could also introduce difficulty in understanding the full I/O layout of a machine by requiring inspection of each EQM individually. In addition, it complicates ownership of I/O signals when such a signal needs to be shared between EQMs. However, this approach could be appropriate for a smaller system, where introducing centralized I/O

handling bloats the internal logic for a few I/O signals.

The template's I/O separation should be done with a dedicated, centralized mapping layer with an I/O handler to enhance the template's modularity and extendability. The handler should be hierarchically owned by the Machine Controller and executed before EQMs are called each cycle. All physical addresses are to be resolved within the handler, and these are passed to the EQMs by the I/O handler. An EQM, therefore, does not concern itself with whether an input originates from physical hardware, a shared signal, or a simulation. This is visualized in Figure 4.9, showing the extra layer of separation that the I/O handler provides. This is a direct application of the Dependency Inversion Principle, where EQMs depend on abstractions through internal variable interfaces, not physical hardware details (Martin, 2008).

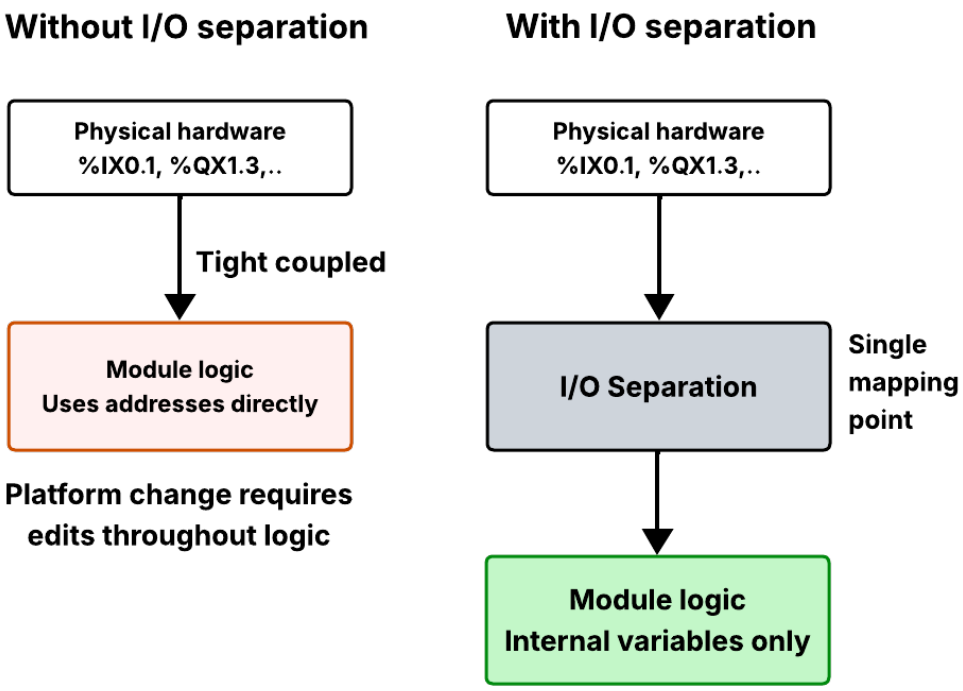


Figure 4.9: I/O handler separation.

#### *4.3.6.2 Simulation Through I/O*

A significant practical consequence of the physical-logical I/O separation is that simulation becomes structurally possible without modification of EQM internal logic. EQMs operate only on inputs of internal variables, and the origin of such inputs can be replaced with a simulation source that mimics real hardware signals. This is derived from test-driven development using mock objects, where a real dependency is replaced with a substitute that mimics the behavior of physical hardware (Freeman and Pryce, 2009).

The template's design of the I/O handler is, in the current design, insufficient to enable full virtual commissioning of a machine; such an integration was out of scope for the thesis. However, with the centralized I/O separation design within the template, it might allow a junior developer to be more inclined toward test-driven development.

## 4.4 Platform Independence

The stated ambition of the framework was to produce an architecture that could serve as a foundation across multiple PLC platforms. In practice, the degree of platform independence achieved requires qualification. The framework will be implemented in Beckhoff's TwinCAT 3, which, among the common industrial PLC platforms, provides broad support for the OOP extensions formalized in the third and fourth editions of IEC 61131-3 (IEC, 2013, 2025). Certain constructs are central to the framework's architecture, specifically inheritance between FBs, abstract methods and interfaces. However, these are not uniformly available on platforms like Siemens TIA Portal or Rockwell Studio 5000, as these systems implement only a portion of the standard's OOP capabilities (Siemens AG, 2026; Rockwell Automation, 2024). This is summarized in table 6.

The platform independence the framework achieves is therefore architectural rather than syntactic. The structural principles it embodies, such as separating state coordination logic from EQM logic, grouping related data into typed structs, expressing EQM lifecycle behavior as explicit state machines and avoiding global variable dependencies, are applicable on any IEC 61131-3 compliant platform. The specific OOP constructs that enforce these principles

in TwinCAT 3, however, would need to be replaced with appropriate equivalents on other target platforms.

The most viable mechanism for achieving this on platforms without inheritance or interface support is composition, described in section 2.3.5. Rather than a derived FB extending a base FB and inheriting its infrastructure, the equivalent behavior can be achieved by declaring an instance of a base FB internally and delegating to it explicitly each scan. Booch (2007) identifies composition as a legitimate and in some respects preferable alternative to inheritance for code reuse, noting that it produces looser coupling since the composing FB is not bound to the internal structure of the component it uses. The alarms infrastructure, initialization pattern and readiness reporting that the base FB provides in the TwinCAT 3 could, in principle, be preserved in this way on platforms that do not support inheritance.

The more significant loss concerns interfaces. Without native interface support, the runtime substitutability that interfaces enable cannot be replicated through language constructs alone. The equivalent on a platform such as TIA Portal would be a standardized FB signature agreed upon by convention: every EQM exposes the same named inputs, outputs and callable blocks, the machine controller interacts with each module through this agreed structure rather than through an interface reference. The architectural separation between coordinator and module is preserved, but the compiler no longer enforces it. The responsibility for maintaining the contract shifts from the tool to the engineer, and must be communicated through documentation and upheld through code review rather than guaranteed at compilation.

Translating the framework to another platform would therefore not be a straightforward syntactic exercise. The data structures are plain IEC 61131-3 types and carry over without modification. The coordinator-module boundary, however, would need to be redesigned from inheritance and interface contracts to composition and convention.

The framework should therefore be understood as a reference implementation in TwinCAT 3, whose architectural intent is transferable even where its specific constructs are not. The principles it applies, and the reasoning behind each design decision, serve as a platform-agnostic design guide for engineers working in environments where the full OOP feature set is unavailable. This provides a conceptual foundation for adaptation to specific target

constraints, as opposed to a codebase intended for direct universal reuse.

Table 6: Platform feature comparison

| Feature | TwinCAT 3 (Beckhoff) | TIA Portal V21 (Siemens) | Studio 5000 V36 (Rockwell) |
|---|---|---|---|
| Composition | Supported | Supported | Supported |
| Encapsulation | Supported | Supported | Supported |
| Inheritance | Supported | Limited | None |
| Interfaces | Supported | Limited | None |
| Methods | Supported | None | None |
| Properties | Supported | None | None |

# 5. RESULT

This chapter presents the implemented template as developed in TwinCAT 3. For each core component, the concrete implementation is described and supported by UML diagrams. Where a component was designed and analyzed but not fully implemented within the scope of the thesis, this is stated explicitly and the design is presented at the conceptual level.

The result is also partly presented through code examples of expected user implementations, where appropriate. This is done using the sorting station code examples from chapter 2, applied to the core components, in their respective section.

## 5.1 Template core components

The template is implemented as a single TwinCAT 3 PLC project with a defined folder hierarchy that separates the reusable template layer from the machine-specific application layer.

The base FBs folder contains the abstract base Function Blocks that form the inheritance foundation of the template: FB_Base, FB_BaseEQM and FB_BaseStateHandler, together with their assorted data type definitions. The Machine Controller folder contains FB_Machine and FB_AlarmHandler, which together form the top layer of every machine built on the framework. The Equipment Modules folder contains the concrete EQM implementations, including the mock modules used for simulation and verification during development. The Reusable FBs folder contains components that are not machine-specific but are not abstract base classes either, including the state handler implementations and the alarm instance FB. The Data Unit Type (DUT) folder contains all shared data type definitions: enumerations, structs and HMI data types. The complete folder structure is illustrated in Appendix A.

The top-level program MAIN contains a single line of logic, instantiating and calling FB_Machine once per scan. All machine logic is encapsulated within FB_Machine, making the Machine Controller fully instantiable and substitutable in accordance with the architectural principles established in section 4.2.

## 5.2 Equipment module base

The equipment module base is implemented as the abstract FB_BaseEQM, which extends FB_Base and implements the interface I_EQMStates. This hierarchy is presented in the class diagram in Figure 5.1, with some attributes and methods omitted for clarity. Every concrete equipment module in the framework must extend FB_BaseEQM and provide implementations for the following abstract methods:

- M_StateMachine
- M_AlarmTriggers
- M_InitAlarms
- M_HMI

These methods are declared as protected and abstract, ensuring that derived modules cannot be instantiated without providing a complete implementation of each. Furthermore, the template implements the following concrete methods:

- M_InitEQM - Sets the module name to its alarm instances.
- M_AlarmIO - Communication with the Alarm Handler.

And the following concrete methods are not yet implemented, but the template is designed using their functionality:

- M_RecipeIO - Communication with the Recipe Handler.
- M_ExchangeIO - Communication with the I/O Handler.

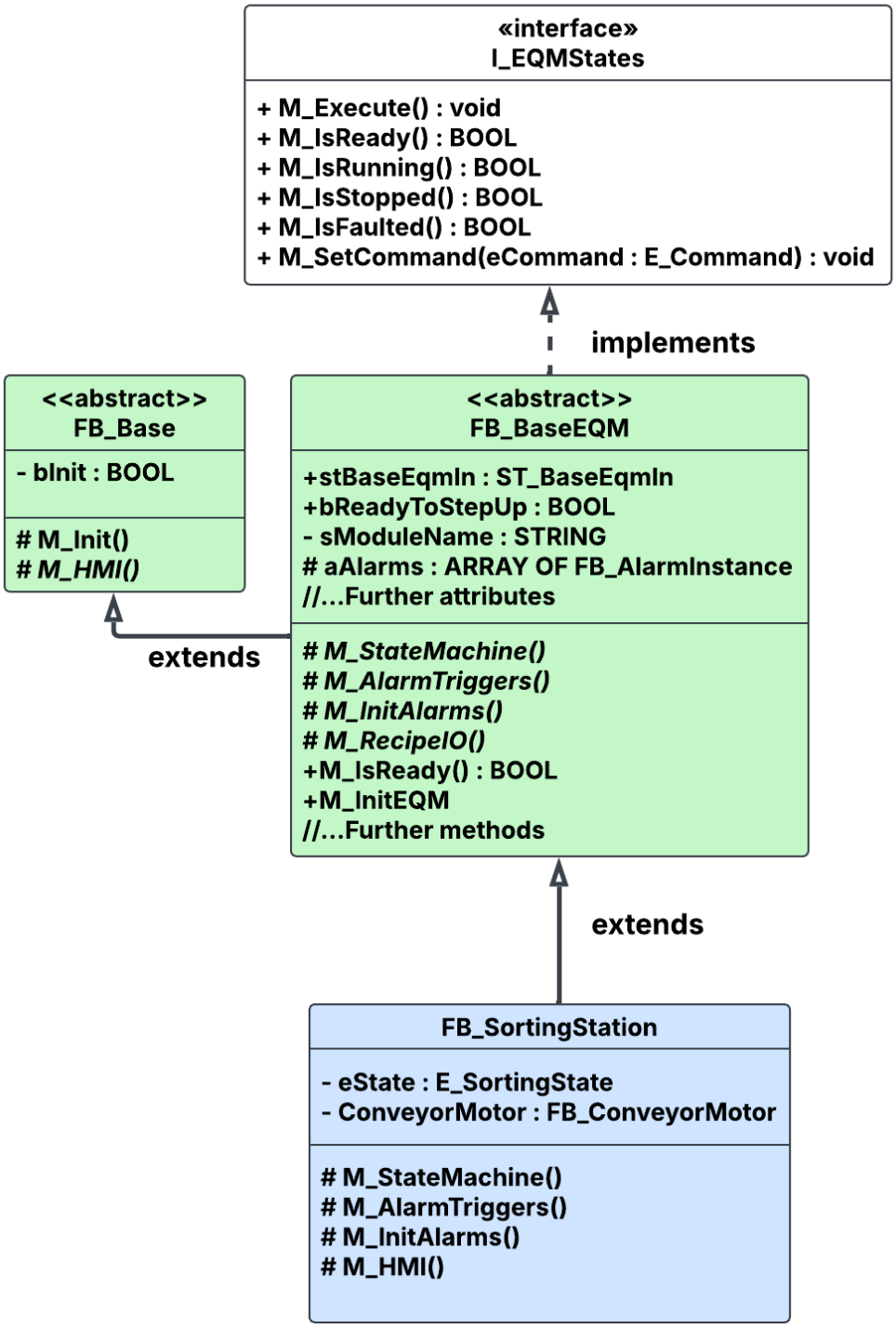


Figure 5.1: Class diagram of SortingStation extending BaseEQM.

The cyclic body of FB_BaseEQM executes the following fixed order each scan: the base initialization check inherited from FB_Base, a one-time EQM initialization block that calls the

methods M_InitEQM() and M_InitAlarms() on the first scan only. Consequent scans execute several methods, as seen in the sequence diagram in Figure 5.2. The bReadyToStepUp output is evaluated as the final operation of each scan by calling M_IsReady().

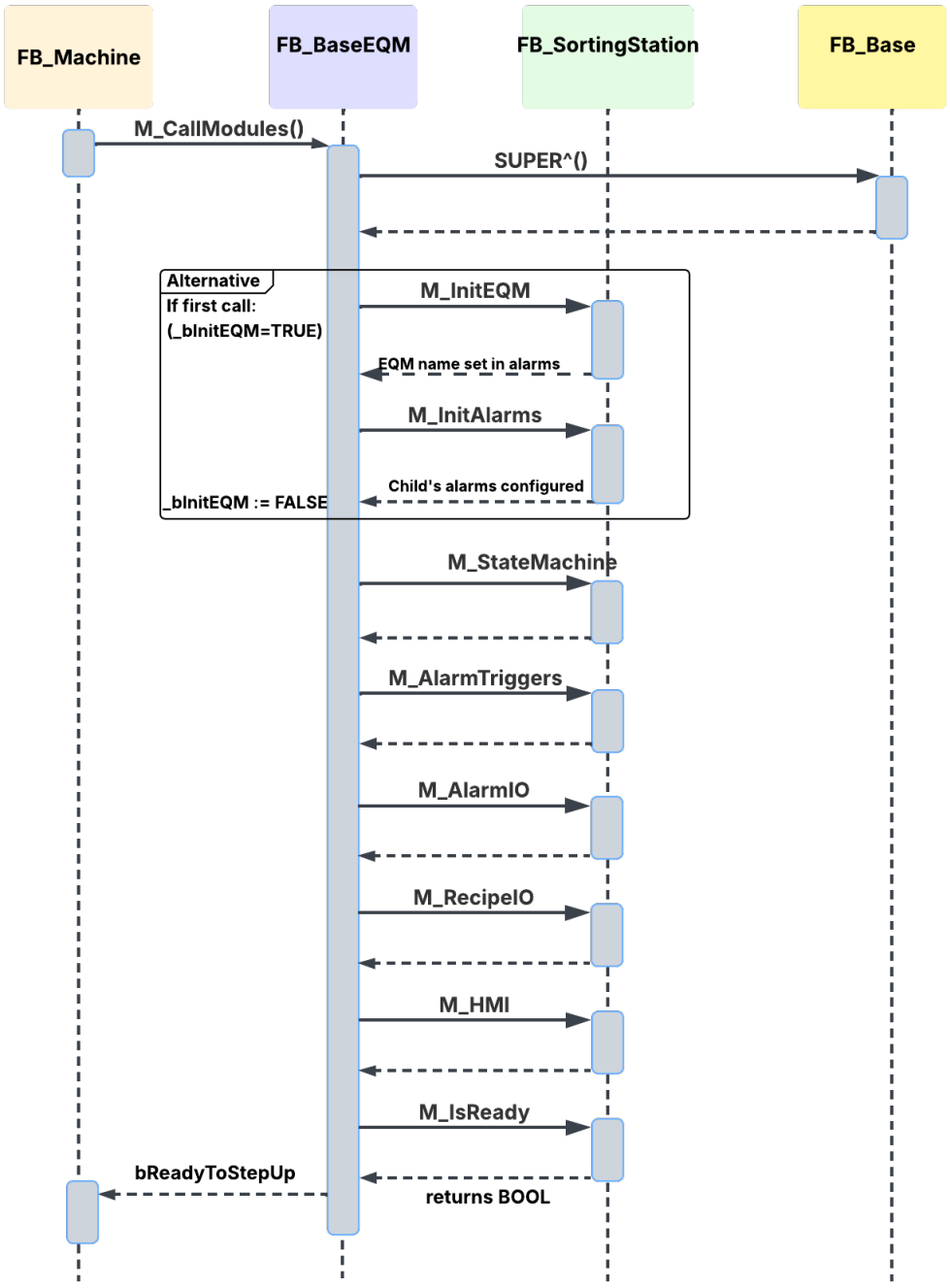


Figure 5.2: Sequence diagram of EQMs.

The interface I_EQMStates defines the complete external contract of any equipment module

as seen by the state coordinator: M_IsReady(), M_IsRunning(), M_IsStopped(), M_IsFaulted(), M_SetCommand(), and M_Execute(). All of these are implemented by FB_BaseEQM, with M_IsReady() provided as an overridable non-abstract method and the status methods left as non-abstract stubs that concrete modules override to map their own internal state variables to the interface contract.

The user of the template is expected to extend FB_BaseEQM as their own implementation of an EQM. Code example 5.1 shows this, using a simplified representation of a sorting station. This shows the overriding of FB_BaseEQM's abstract method M_StateMachine. The EQM methods related to alarms are presented in section 5.4.

```
FUNCTION_BLOCK FB_SortingStation EXTENDS FB_BaseEQM
VAR
    eSortingState : E_SortingState;
    ConveyorMotor : FB_ConveyorMotor;
    Scanner : FB_BarcodeScanner;
    Pusher: FB_Pusher;
END_VAR
VAR_INPUT
        stBaseEqmIn : ST_BaseEqmIn // Broadcast from FB_Machine
END_VAR
METHOD PROTECTED M_StateMachine
CASE stBaseEqmIn.eMachineStates OF
    E_MachineStates.Idle:
        // Waiting for coordinator START command
    E_MachineStates.StartUp:
       //StartUp sequence
    E_MachineStates.Production:
       CASE eSortingState OF
        E_SortingState.ConveyorRunning:
         IF PackageDetected THEN
            ConveyorMotor.Stop();
            eSortingState := E_SortingState.PackageDetected;
         END_IF;
        E_SortingState.PackageDetected:
        //Scan Barcode, transition to PushItem
        eSortingState := E_SortingState.PushItem;
        //... Further logic for sorting station statemachine
    E_MachineStates.Shutdown :
        // Sorting station Shutdown sequence
```

Code Example 5.1 – Sorting station implemented as an EQM.

## 5.3 State coordinator

The State Coordinator is implemented across two layers. FB_BaseStateHandler provides the abstract coordination infrastructure, extending FB_Base and implementing I_StateHandler. This architecture is visualized in Figure 5.3, in a class diagram including an example of a user implementation of FB_StatesSortingStation.

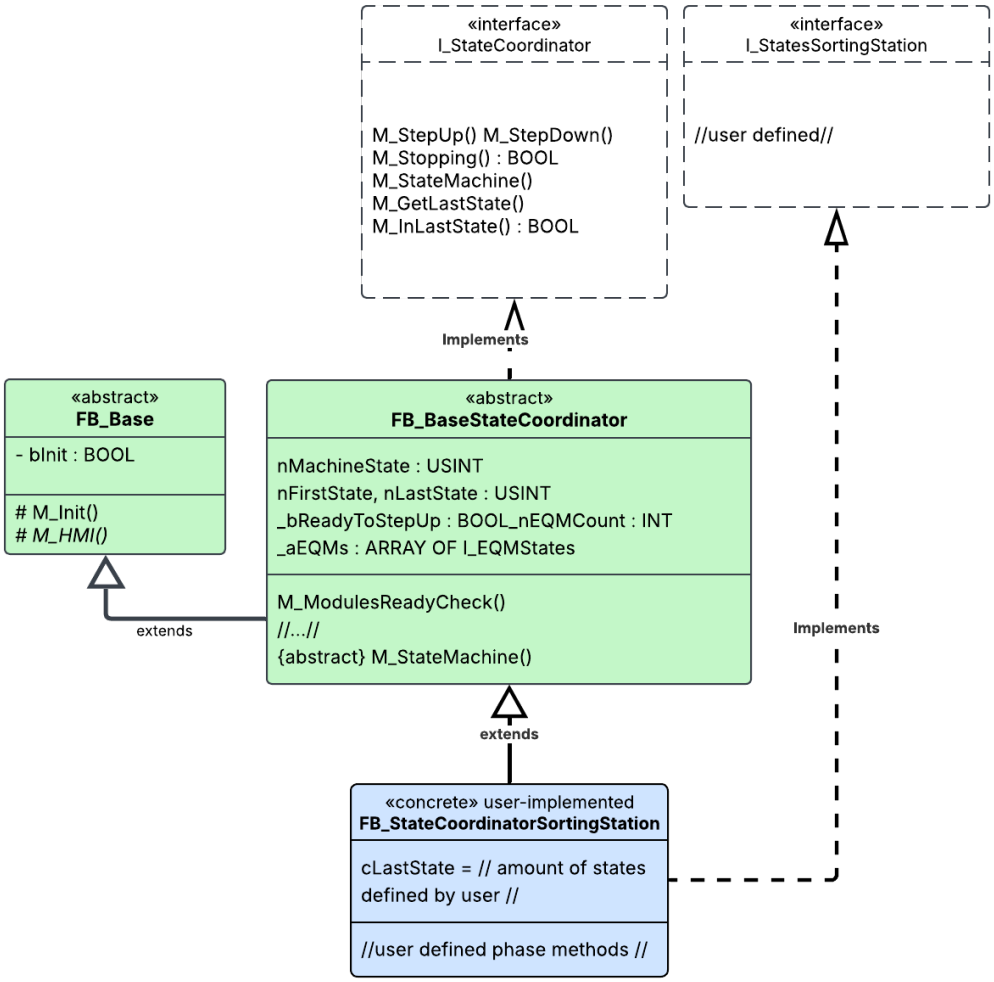


Figure 5.3: Class diagram of State Coordinator.

Implementations of a State Coordinator extend FB_BaseStateHandler and provide machine-specific phase sequence logic through the abstract M_StateMachine() method. In each cycle, FB_Machine calls the State Coordinator, which executes two main methods. First, M_CheckModulesReady() is called, collecting the report of all EQMs, whether they are ready to transition into the next machine state; if all of them are ready, this is broadcast in

the next cycle. This collection and reporting sequence is visualized in Figure 5.4 below.

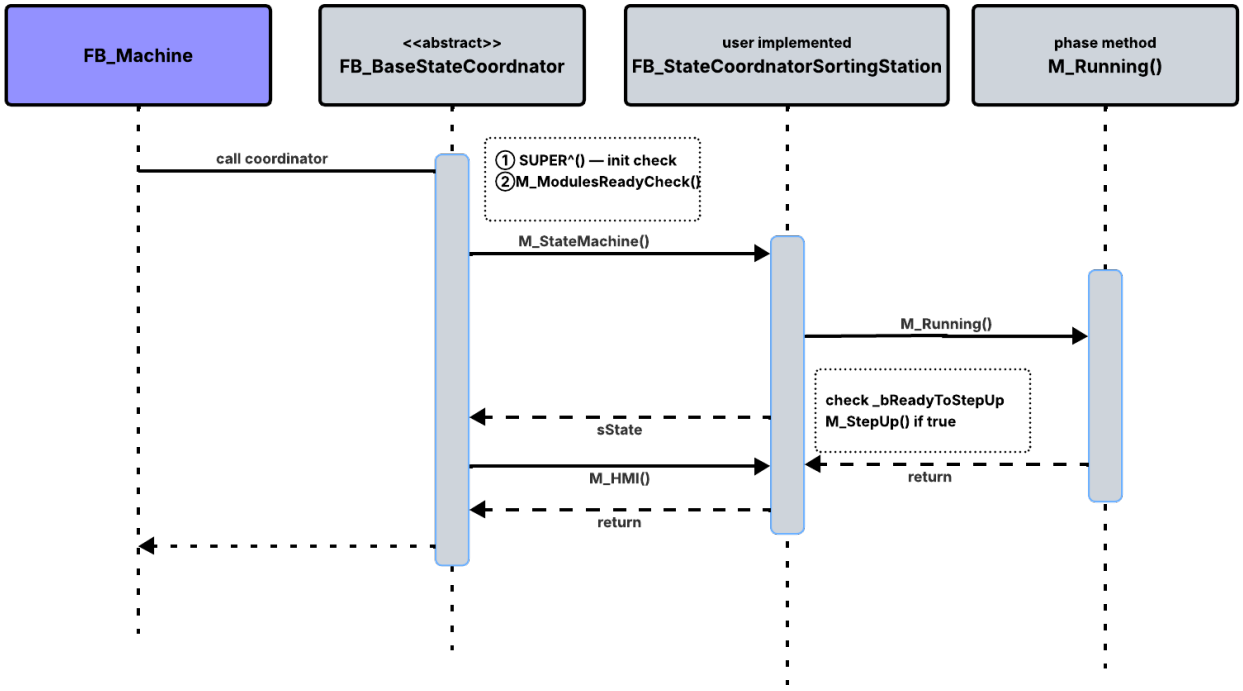


Figure 5.4: Sequence diagram of State Coordinator.

Secondly, the State Coordinator runs a user-implemented M_StateMachine(), which is expected to govern the machine as a whole. Code example 5.2 shows how the State Coordinator of a sorting station could be implemented using the template.

```
FUNCTION_BLOCK FB_StateCoordinatorSortingStation EXTENDS
   ↪ FB_BaseStateCoordinator
VAR CONSTANT
    cLastState : USINT := 3;
END_VAR
METHOD PROTECTED M_StateMachine
CASE nMachineState OF
    0: M_0_Idle();
    1: M_1_Startup();
    2: M_2_Production();
    3: M_3_Shutdown();
END_CASE;
END_METHOD
METHOD PRIVATE M_0_Idle
    IF _bReadyToStepUp THEN
      M_StepUp();
    END_IF;
END_METHOD
METHOD PRIVATE M_1_Startup
    M_BroadcastCommand(E_EQMCommand.eSTART);
    IF _bReadyToStepUp THEN
        M_StepUp();
    END_IF;
END_METHOD
METHOD PRIVATE M_2_Production
    IF (E_EQMCommand.eSTOP) THEN
       M_StepUp();
    END_IF;
END_METHOD
METHOD PRIVATE M_3_Shutdown
    IF M_Stopping() THEN
       M_StepDown();
    END_IF;
END_METHOD
```

Code Example 5.2 – Implemented State Coordinator for a sorting station.

## 5.4 Alarm handler

The handling of alarms within the template is presented in this section. To begin with, the lifecycle of alarms inside EQMs as objects of FB_AlarmInstance; how they are initialized, triggered, maintained internally and interface with the alarm handler. Then, a brief description of the internal logic and functionality of the alarm handler as an implemented object, FB_AlarmHandler.

FB_AlarmInstance, seen in code example 5.3, is an FB that manages the lifecycle of a single alarm within an EQM. It has, among other variables, an ST_Event previously described in chapter 4.3.3. Furthermore, each alarm also has executable code inside its implementation section, related to that alarm. This executes during every cycle and is responsible for tracking whether its alarm condition is currently active, whether it has been acknowledged by an operator, and when it first occurred.

```
1  FUNCTION_BLOCK FB_AlarmInstance
2  VAR_INPUT
3     stEventInit : ST_Event;
4     bTrigger : BOOL;
5     bAckAlarm : BOOL;
6     bDisable  : BOOL;
7  END_VAR
8  VAR_OUTPUT
9     stEvent : ST_Event;
10    END_VAR
11 VAR
12    fbAlarmTriggered : R_TRIG;
13    bInit : BOOL := TRUE;
14 END_VAR
15    //Body of FB_AlarmInstance, executed cyclically for each
         ↪ alarm
16    //Watches the trigger condition for an alarm and if an
         ↪ alarm has been acknowledged
17 END_FUNCTION_BLOCK
```

Code Example 5.3 – FB_AlarmInstance.

An array of FB_AlarmInstance objects exists within EQMs, representing all alarms that an EQM has. Figure 5.5 visualizes how, during each scan cycle, all alarms are interacted with for all EQMs. The EQMs' method M_AlarmIO iteratively calls each alarm in the array of FB_AlarmInstance. The output from each FB_AlarmInstance, stEvent, is copied to an array of ST_Events within each EQM. FB_AlarmHandler receives this array as an input, using the Machine Controller FB_Machine as an intermediary.

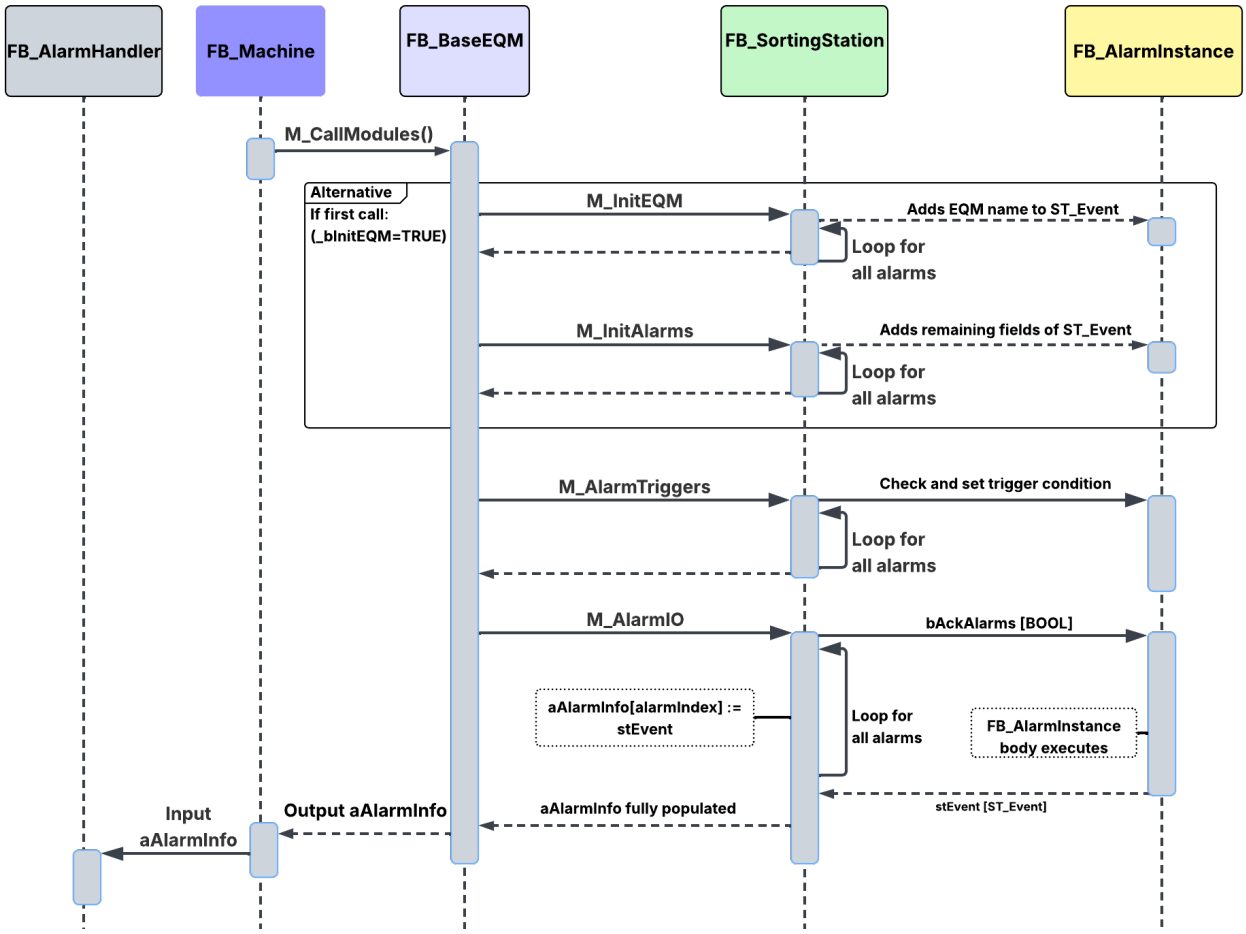


Figure 5.5: Sequence diagram of alarm lifecycle within EQMs.

A user would minimally need to implement the two methods M_InitAlarms and M_AlarmTriggers. An example of this, continuing the sorting station example, is illustrated below in code example 5.4. This implements an alarm that would be raised if an overcurrent in the conveyor motor is detected.

```
METHOD PROTECTED M_InitAlarms
VAR
END_VAR
_aAlarms[1].stEventInit.nID := 1;
_aAlarms[1].stEventInit.eStateRequest:=E_MachineStates.
   ↪ PowerOff;
_aAlarms[1].stEventInit.eStopRequest:=E_StopTypes.FastStop;
_aAlarms[1].stEventInit.sText:='Conveyor motor overcurrent
   ↪ fault';
_aAlarms[1].stEventInit.eSeverity:= TcEventSeverity.Critical;
//...further alarms
END_METHOD
METHOD PROTECTED M_AlarmTriggers
VAR
END_VAR
_aAlarms[1].bTrigger := bMotorOvercurrentDetected
//...further alarms
END_METHOD
```

Code Example 5.4 – Sorting station as EQM implemented alarm methods.

FB_Alarmhandler is implemented as a concrete FB extending FB_Base. It is called by FB_Machine every scan cycle and has three main jobs: tracking which alarms are active, managing acknowledgement, and outputting data to an HMI.

It receives an alarm array from FB_Machine as a VAR_IN_OUT and processes it each scan through M_CheckAlarms(). It maintains two internal lists: an active alarm list and a history list, both sized according to constants in GVL_Constants. The alarm handler exposes stop request information to the Machine Controller, FB_Machine, as a boolean array indexed by E_StopTypes, one flag per stop type. This allows FB_Machine to evaluate which stop types are active and resolve priority without requiring direct access to individual alarm instances.

## 5.5 Recipe handler

The recipe handler was designed but not fully implemented in TwinCAT 3 within the scope of the thesis. The design is partly presented here as a specification for future implementation. As previously described in chapter 4.3.4, the responsibility of the handler is orchestration of

the recipe lifecycle from operator selection, parsing, validation and runtime monitoring. It acts through the state machine established in chapter 4.3.4. The designed class architecture of the recipe handler is visualized in Figure 5.6.

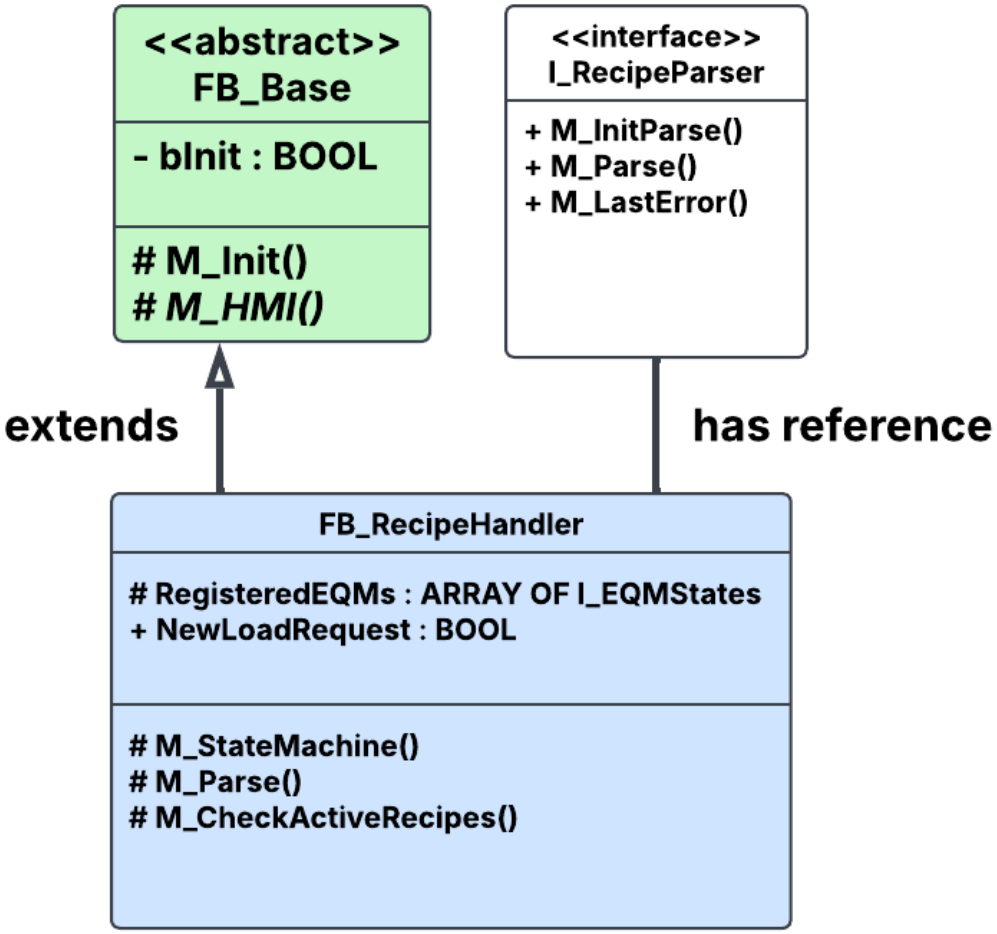


Figure 5.6: Partially implemented RecipeHandler class diagram.

The designed, but not implemented, approach of how recipes should be implemented is similar to that of FB_AlarmInstance, previously presented in 5.4, as an FB_Recipe. This would hold ST_BaseRecipe as an attribute and contain the logic to communicate with FB_RecipeHandler. The user of the template would need to implement EQM-specific structs, as well as the 'metadata' of such a struct. How such an implementation could look is presented in code example 5.5, which is an extension of the previously presented recipe struct in chapter 2.6.4. The recipe handler would use the 'metadata'-array for the correct parsing of different filetypes and populating the EQM-specific recipe structs.

```
TYPE ST_SortingStationRecipe :
STRUCT
    MotorSpeed : REAL;
    LaneBarcodePrefix : ARRAY[1..10] OF STRING(32);
    PusherExtendDelay   : TIME;
END_STRUCT
END_TYPE
FUNCTION_BLOCK FB_RecipeInstance
VAR_INPUT
END_VAR
VAR_OUTPUT
VAR
   bInit : BOOL := TRUE;
   stBaseRecipe : ST_BaseRecipe;
    //added by user
   stSortingStationRecipe : ST_SortingStationRecipe;
END_VAR
VAR CONSTANT
     //added by user
    sMetaData : ARRAY[1..4] OF STRING(64) := [
        'MotorSpeed:REAL',
        'LaneBarcodePrefix:ARRAY[1..10] OF STRING',
        'PusherExtendDelay:TIME',
        'PusherExtendTime:TIME'
         ];
END_VAR
END_FUNCTION_BLOCK
```

Code Example 5.5 – Example of user-defined sorting recipe.

To allow the recipe handler to function irrespective of which file type or storage the user has decided to use, it has a reference to the interface I_RecipeParser. FBs such as FB_XMLParser or FB_CSVParser should implement this interface.

## 5.6 I/O handler

The I/O Handler was designed and analyzed as described in section 4.3.6, but was not implemented in TwinCAT 3, within the scope of this thesis. The architectural decisions are documented in chapter 4.3.6. This section aims to show the conceptual application of the design and architectural decisions. Figure 5.7 visualises the class architecture of the I/O Handler.

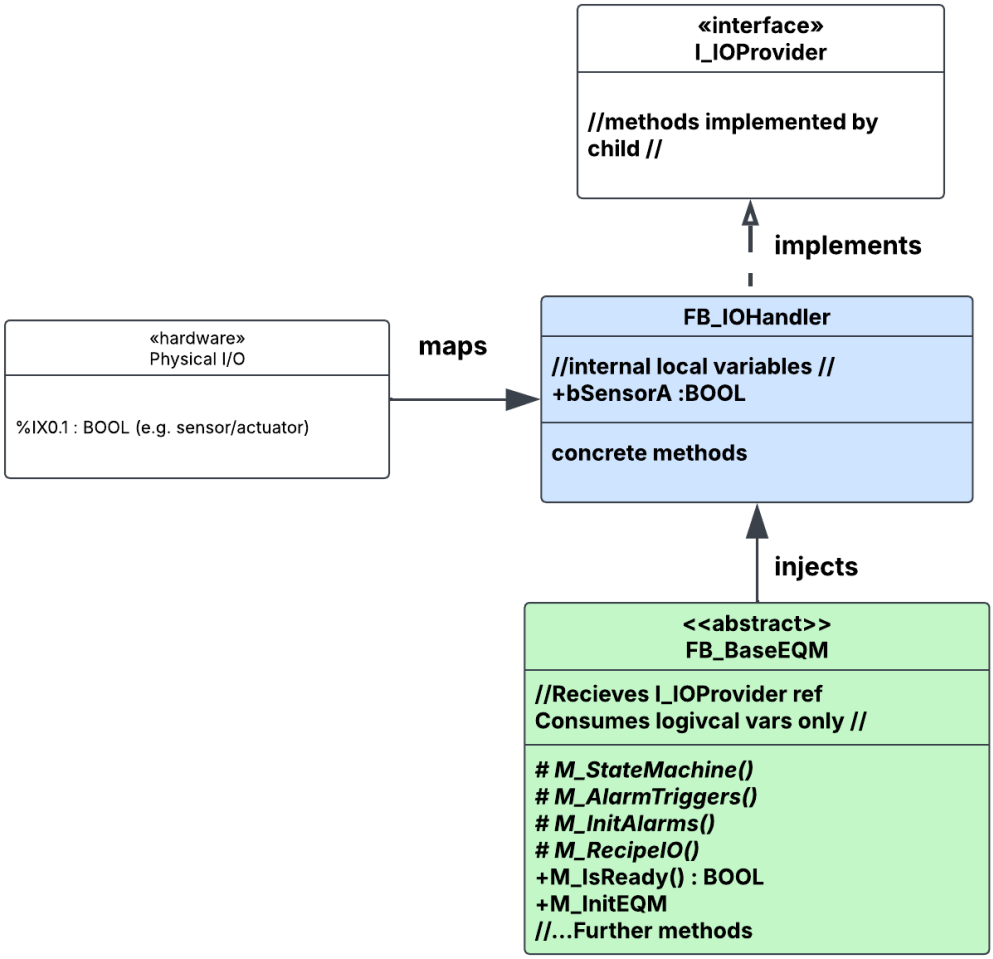


Figure 5.7: Class architecture of I/O Handler.

The I/O Handler is an FB that acts as the single intermediate gateway between physical I/O signals and the EQMs. It utilizes user-defined input and output structs; a simplified example of these is presented in code example 5.6.

```
TYPE ST_SortingStationInput :
STRUCT
     MotorFaultSignal : BOOL;
     //...further inputs for a sorting station
END_STRUCT
END_TYPE
TYPE ST_SortingStationOutput :
STRUCT
     MotorRunSignal : BOOL;
     //...further outputs for a sorting station
END_STRUCT
END_TYPE
```

Code Example 5.6 – User-defined Input/Output structs.

At the start of each cycle, the I/O Handler reads all external inputs and packages them into the input structs. These structs are used as inputs during FB_Machine's method call M_CallModules to all EQMs. The EQMs execute using the input struct data, and before finishing, the EQMs populate their output structs. Before finishing the scan cycle, the I/O Handler receives the collection of output structs, and writes the results to physical outputs. Figure 5.8 illustrates this sequence for a sorting station, using one input and one output.

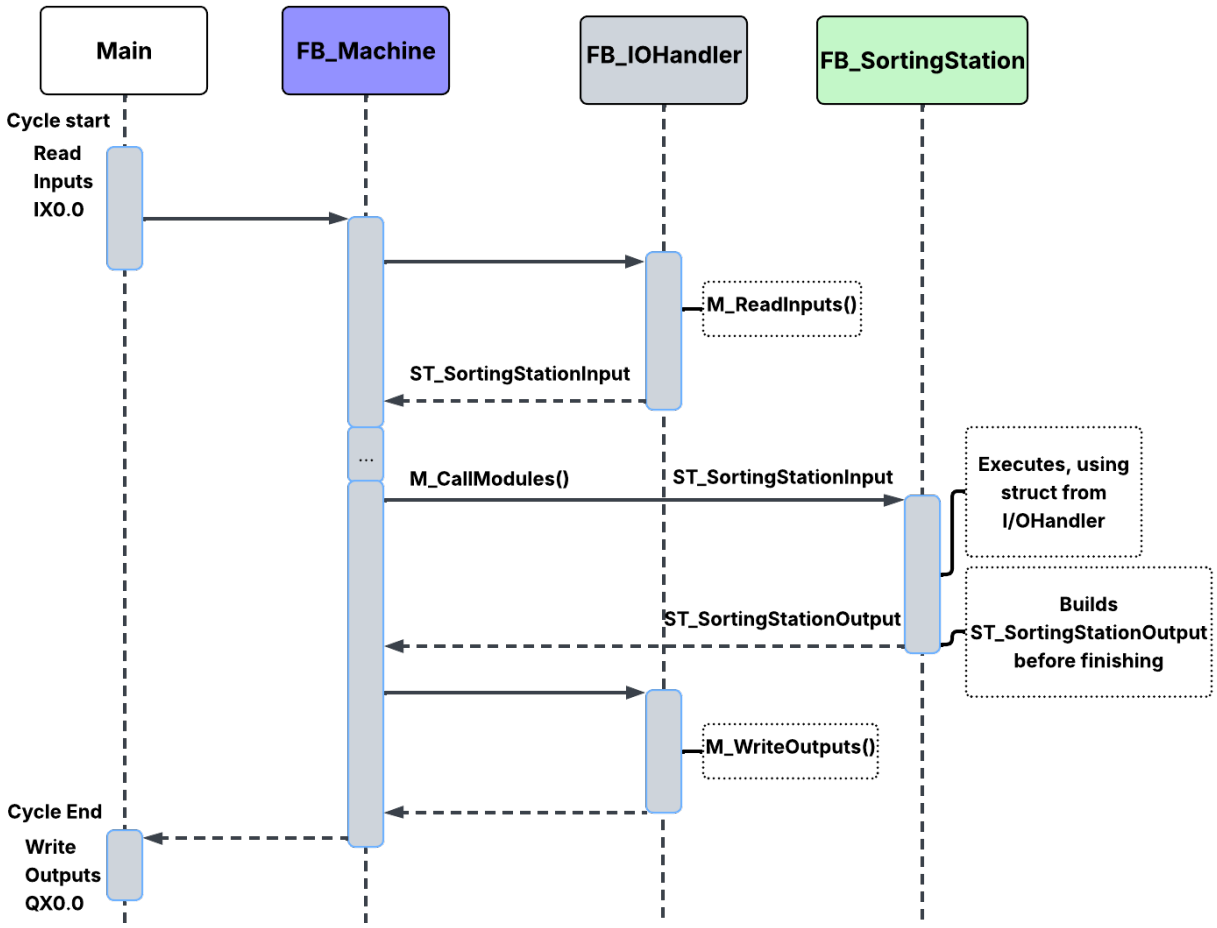


Figure 5.8: Sequence diagram of I/O Handler.

## 5.7 Safety Integration / Stop & Recovery Handling

The safety integration with stop & recovery handling was designed and analyzed as described in section 4.3.5. A fully integrated implementation, which would allow for the connection of a dedicated safety PLC, was not completed. However, some functionality was integrated into the template via FB_Safety, a concrete EQM extending FB_BaseEQM. It participates in the readiness gate, configures alarm instances through M_InitAlarms, and exposes the public abstract method M_IsSafetyOK. FB_Machine calls this method each scan when assembling the broadcast struct. In the current implementation, the method returns a configurable internal boolean, serving as a placeholder for physical safety hardware connections as described in section 2.6.5.

## 5.8 Machine Controller

FB_Machine is the top-level machine controller. It extends FB_Base and owns instances of the State Coordinator, alarm handler, recipe handler, I/O Handler, all EQMs, and the HMI data structure. Its cyclic body calls M_CallModules, flattens alarm arrays into a single flat array, calls the alarm handler, and updates HMI output through M_HMI. The broadcast struct is assembled at the start of M_CallModules, before any module is called, ensuring all modules receive identical machine context within the same scan. Stop type priority is resolved in the same method using a conditional chain evaluating the alarm handler's stop request array, writing the highest-priority active stop type to the broadcast struct.

```
FUNCTION_BLOCK FB_Machine EXTENDS FB_Base
VAR
//  State Coordinator
    fbStateCoordinator: FB_StateCoordinatorSortingStation;
//  Handlers
    fbAlarmHandler : FB_AlarmHandler;
//  Broadcast struct
    stBaseEqmIn : ST_BaseEqmIn;
//  Equipment modules
    fbSortingStation: FB_SortingStation;
    fbSafety : FB_Safety;
//  Alarm collection
    //Alarm array, mediated between EQMs and AlarmHandler
// To be implemented/integrated
    fbRecipeHandler : FB_RecipeHandler;
    fbIOHandler : FB_IOHandler;
END_VAR
SUPER^()
// 1. Update broadcast struct
    // Current Machine state from fbStateCoordinator
    // Acknowledgements of alarms from Alarmhandler
    // Stop request from Alarmhandler
    // Safety signal from SafetyHandler
// 2. Run State coordinator
    fbStateCoordinator();
// 3. Run all EQMs
    M_CallModules();
// 4. Run alarm handler
    fbAlarmHandler(); //INPUT/OUTPUT alarm array
// 5. Call HMI
    M_HMI()
METHOD PRIVATE M_CallModules
    //Logic to safely call all modules
       fbSortingStation(); //INPUT: broadcast struct
METHOD PROTECTED M_Init
    // Registers EQMs, held in an array of I_EQMstates,
       ↪ forwards to State Coordinator
METHOD PROTECTED M_HMI
    // populates a machine HMI struct
```

Code Example 5.7 – FB_Machine as the top level machine controller.

The first step assembles the broadcast struct ST_BaseEqmIn. This is done before any module is called, ensuring all modules receive identical machine context within the same scan. The current machine state, safety status, active stop request, and alarm acknowledgement flag are all written to the struct at this point. Stop type priority is resolved here using a conditional

chain that evaluates the alarm handler's stop request array and writes the highest priority active stop type to the struct.

The second step calls the state coordinator, passing the readiness array and stop request information. The coordinator runs its own body, evaluates module readiness, and advances or retreats the machine state accordingly.

The third step calls all equipment modules through M_CallModules. Each module receives the broadcast struct as a VAR_INPUT and returns its readiness flag and alarm output array via explicit output wiring at the call site.

The fourth step calls the alarm handler, passing the flat alarm array. The alarm handler processes active alarms, resolves the outgoing stop request array, and populates its HMI output struct.

The fifth and final step, the M_HMI method is called to drive the specific terminal or HMI device implemented by the user.

FB_Machine is itself a Function Block rather than a Program, which means it can be instantiated inside a test harness and its modules replaced with mock implementations through their interface contracts. This is the structural property that makes desktop-level validation possible without physical hardware, as discussed in section 4.2.1.

# 6. DISCUSSION & CONCLUSION

This chapter summarizes the findings of the thesis, discusses the results, and reflects on the work as a whole. It aims to provide answers to the research questions, discuss ethical considerations and directions for further development of the framework.

## 6.1 Discussion

The scope of the thesis proposal introduces a level of structural complexity that became increasingly evident throughout the different phases of the work. The six core components defined in section 1.1 each represent a significant design challenge in their own right. This perspective was reinforced by both the empirical study and interviews, each associated with established design patterns. The codebase review also highlighted the diversity of practical implementations across systems.

The decision to address all six components within a single framework, and to analyze each in relation to both the literature and the interview findings, meant that the analysis layer of the thesis was extensively broad. This was necessary for establishing the architectural foundations of the framework, but also revealed that each component influences the design of the others, meaning that each component had to be understood before any could be implemented definitively.

Despite this, the framework as delivered addresses the research questions coherently and consistently. The implemented components form a working foundation that can be extended with machine-specific logic.

The framework is, in its current state, most appropriate for machines of moderate to high complexity, where multiple distinct subsystems must be coordinated and where the codebase will be maintained, extended, or handed over across engineers and time. The structured hierarchy, typed communication, and compiler-enforced contracts provide the greatest return in precisely these conditions. Projects where several engineers work in parallel or where onboarding time is a real cost benefit directly from the predictable structure the framework imposes.

The framework imposes a fixed structural cost on every project. Every EQM must implement seven abstract methods, regardless of how simple its logic is. The inheritance hierarchy, broadcast struct, alarm infrastructure, and coordinator registration are all present from the moment a project is created. For a very simple machine, such as a single actuator with one sensor and two alarms, this overhead is disproportionate: the engineer writes more infrastructure than application logic. A flat Function Block with direct I/O wiring would be more appropriate and easier to maintain in such a context. It is therefore questionable if the framework is well suited to single-module or very simple low-count component machines where the structural overhead outweighs the organizational benefit.

The framework also assumes that the machine architecture translates naturally into discrete Equipment Modules. A program without clear module boundaries may not map well onto the EQM structure. Forcing such a program into the framework risks producing an architecture that is formally compliant but structurally artificial.

Finally, the OOP constructs the framework depends on are not uniformly available across all PLC platforms. On platforms with limited support for abstract methods and interfaces, the compiler-enforced contracts would need to be replaced by convention and documentation, reducing the structural guarantees the framework is designed to provide.

The framework is best understood as a starting point for a specific class of problems, not a universal solution. Engineers adopting it should assess whether the machine architecture maps naturally onto the EQM model and whether the project scope justifies the structural investment.

### *6.1.1 Hardware Independence*

The OOP constructs used in the framework are language-level features resolved by the compiler before any code reaches the PLC. The hardware executes compiled instructions and has no awareness of the source structure that produced them (Antonsen, 2020). The framework therefore imposes no hardware requirements beyond those of any standard TwinCAT 3 project: sufficient processing capacity and memory to run the program within the configured cycle time.

This changes when machine components carry their own processing intelligence and communicate through industrial protocols such as EtherCAT, PROFINET, or EtherNet/IP. Devices such as robot controllers, servo drives, or safety PLCs expose data models, parameter structures, and device-specific state machines through these protocols (Kurose and Ross, 2012). The framework does not eliminate the engineering effort required to integrate such devices, nor is it intended to. That effort is inherent in the device and its protocol.

What the framework provides is a defined location for that complexity. All device-specific communication logic is encapsulated within the relevant EQM and its I/O mapping layer, as discussed in section 4.3.5. The state coordinator and the alarm handler remain insulated from the details of any particular device interface. They interact only with the I_EQMStates contract, regardless of whether the underlying module communicates over a fieldbus or reads from a digital terminal.

## 6.2 Research Questions

This section provides concise answers to the seven research questions stated in section 1.4, based on the findings in the literature review, the empirical study and the developed framework.

**How are PLC programs typically structured in the industry today?**

Industry practice converges on a modular, hierarchical folder structure where each subsystem occupies its own program or code block, with routines beneath it for individual functions. Naming conventions aligned with electrical schematics and inline comments explaining design intent are universally recognized as the most impactful individual quality practices. However, in practice, codebase age and the absence of enforced standards produce significant structural variation between engineers and projects, with legacy codebases representing the most severe case of accumulated inconsistency.

**What is required for the framework to be platform-independent?** Full syntactic platform independence is not achievable with a framework grounded in OOP constructs, as these are not uniformly supported across IEC 61131-3 compliant platforms. The platform

independence the framework achieves is architectural: the structural principles it embodies are applicable on any compliant platform, even where the specific language constructs enforcing them must be adapted. The framework should therefore be understood as a reference implementation whose architectural intent is transferable, rather than a codebase intended for direct reuse across all platforms.

**What problems arise during maintenance and troubleshooting?** The primary barriers are poor or ambiguous naming, missing or superficial inline documentation, and structural inconsistency between developers. These problems compound in legacy codebases where the original intent is no longer recoverable. Tight coupling between control logic and physical hardware was also identified as a significant barrier to desk-level debugging. The absence of a functional specification before implementation was identified as a root cause of downstream maintenance and testing problems.

**How can a general PLC architecture be designed for scalability and reusability?** Scalability and reusability are achieved by enforcing consistent structural boundaries between components, keeping shared infrastructure in a single location, and making inter-component dependencies explicit and typed. A framework that encodes these properties structurally, rather than relying on convention and documentation, reduces the effort required to extend, adapt, and onboard new engineers into any given project built on it.

**How can object-oriented principles be applied to PLC programming in accordance with IEC 61131-3?**

The fourth edition of IEC 61131-3 provides direct support for encapsulation, inheritance, and polymorphism. These constructs are fully applicable in TwinCAT 3 and were used as the primary mechanism for enforcing structural contracts in the framework. Their application must, however, be disciplined: inheritance depth should be kept shallow, and interface contracts should be designed with cross-platform constraints in mind to preserve portability at the architectural level.

**How should core components be managed in a standardized framework?**

Each core component should address a single, distinct concern and communicate with sur-

rounding components through well-defined constraints rather than direct dependencies. Centralizing responsibilities such as alarm management, state coordination and I/O mapping into dedicated components, while distributing module-specific behavior to the modules themselves, produces a system where individual components can be modified, extended, or replaced without cascading changes to the rest of the architecture.

**How can the solutions be tested and simulated?**

Hardware-independent testability is most effectively achieved as a structural property of the architecture rather than a separate testing layer. When components communicate through typed interfaces and structs, any component can be replaced by a simulated equivalent without modifying surrounding logic. The extent to which this can be realized in practice is, however, partly platform-dependent as the availability of virtual PLC environments and software-level simulation tools varies significantly between platforms. TwinCAT 3, as the primary implementation target of this framework, provides built-in simulation capabilities that make desktop-level validation feasible without physical hardware. While TwinCAT 3 is not the only platform to support this, it does so without licensing fees.

## 6.3 Ethical Considerations

The empirical study conducted in this thesis involved access to two categories of sensitive material: internal source code from previous Etteplan projects, and personal data collected through recorded interviews with engineers.

### *6.3.1 Confidentiality and Handling of Confidential Information*

The source code reviewed during the pre-study was provided by Etteplan for the purpose of informing the framework design. This material was treated as confidential throughout the project. No proprietary code has been reproduced or disclosed in this report, and the analysis derived from it is presented only at the level of structural patterns and architectural observations, without reference to specific client projects or identifiable implementations.

The interviews were conducted with the informed consent of each participant. Before each interview, respondents were informed of the purpose of the study, how the recorded material

would be used, and that participation was voluntary. Consent for audio recording was obtained explicitly. In accordance with the GDPR, interview recordings and transcripts were stored securely and accessed only by the authors of this thesis, and used solely for the analytical purposes described in the method chapter. Furthermore, respondents are not identified by name in this report and only referred to by coded identifiers throughout the analysis.

## 6.4 Future Development

The most immediate continuation is completing the components designed and analyzed but not implemented within the available time-frame, specifically the recipe handler and I/O handler. Both are documented at the design and specification level in sections 4.3.4 and 4.3.6, and the implemented components are structurally prepared to receive them.

Deployment in a real Etteplan project would provide a substantially stronger basis for assessing practical utility. Requirements and edge cases that are difficult to anticipate during design tend to surface under production conditions, and iterative refinement under real project constraints is the natural next phase of framework development.

A formal inter-machine communication mechanism would extend the framework's applicability to larger production lines and multi-zone machines. The instantiable nature of FB_Machine already allows multiple independent machine instances to run within a single PLC, each governing its own Equipment Modules and state coordinator. Building on this, a dedicated inter-machine struct, analogous to ST_BaseEqmIn, could carry zone-level readiness and state information between instances in a controlled and explicit way, without compromising the unidirectional communication principles that govern the single-machine architecture.

Expanding platform support is a further natural direction. Producing platform-specific companion implementations for TIA Portal and Studio 5000, using composition and convention-based signatures where inheritance is unavailable, would bring the platform-independent architectural ambition closer to realization, as discussed in section 4.4.

## 6.5 Conclusion

The framework developed in this thesis provides Etteplan with a structured, modular starting point for PLC development grounded in the fourth edition of IEC 61131-3 and validated against industrial practice through five semi-structured interviews and a codebase review. Its core contribution is moving structural consistency from convention to enforcement: abstract method contracts, typed inter-component communication, and a compiler-verified module readiness gate ensure that projects built on the framework conform to its architecture regardless of who writes the application logic.

The six core components identified in section 1.1 are addressed in the analysis and either implemented or fully specified for continuation. The result directly serves the purpose stated in section 1.2: a reusable foundation that improves code quality and readability, reduces redundant reimplementation across projects, and provides junior and senior engineers alike with a navigable and structured starting point for future PLC projects.

# TERMINOLOGY

This chapter defines technical terms used throughout the thesis that are not assumed to be part of the prior knowledge of a reader with a background in computer engineering or electrical engineering. The terms are alphabetically listed.

**Abstract Method:** A method declared within a Function Block without a corresponding implementation. Any Function Block that extends an abstract base is required by the compiler to provide a concrete implementation of each abstract method. In the context of IEC 61131-3 and TwinCAT 3, abstract methods serve as enforceable contracts that define what behavior a derived Function Block must provide, without prescribing how that behavior is realized.

**Commissioning:** The process of bringing a newly developed or modified automation system into operation on physical hardware. During commissioning, program logic that has been developed and validated in a simulation or test environment is deployed to the target controller and verified against real inputs and outputs from the machine. Commissioning typically represents the final stage before a system is handed over for production use.

**Cyclic Scan:** The repeating execution loop that governs how a PLC processes its program. In each scan cycle, the controller reads physical input signals, executes all program logic in sequence, and writes the resulting output values to the output hardware. This cycle repeats continuously at a fixed or near-fixed interval, ensuring that the program responds to changes in the controlled process within a predictable and bounded time.

**Equipment Module (EQM)** A self-contained software unit responsible for encapsulating the logic of one mechanical or functional section of a machine. Each Equipment Module manages its own internal state, alarm conditions, and HMI data, and communicates its readiness and operational status to the machine coordinator through a defined interface. The term originates in the ISA-88 batch control standard, though it is applied in this thesis in a broader sense to any modular subsystem within the framework.

**Function Block (FB)** A Program Organization Unit defined in the IEC 61131-3 standard

that encapsulates logic together with persistent internal state variables. Unlike a Function, which is stateless, a Function Block retains its state between successive calls, making it suitable for cyclic logic such as timers, counters, and device control sequences. Multiple independent instances of a Function Block may be created within a project, each maintaining its own set of internal variables. In the third and fourth editions of IEC 61131-3, Function Blocks also support methods, inheritance, and interface implementation, enabling object-oriented design patterns.

**Human-Machine Interface (HMI):** The operator-facing layer of an automation system through which process values are displayed and control commands are issued. In the context of a PLC system, an HMI typically communicates with the controller over an industrial network, reading status variables and writing setpoints and commands. In the framework developed in this thesis, each Function Block exposes a dedicated HMI data structure that aggregates the information relevant to its section of the machine.

**IEC 61131-3** An international standard published by the International Electrotechnical Commission that defines a unified software model and a set of programming languages for programmable logic controllers. The standard specifies five programming languages, including Structured Text and Ladder Diagram, as well as the Program Organization Unit model that governs how software elements are structured and interact. Successive editions of the standard have progressively extended support for object-oriented programming constructs, with the fourth edition formalizing interfaces, abstract methods and inheritance.

**Input/Output (I/O) Handling:** The part of a PLC program responsible for managing the connection between the program's internal logic variables and the physical signals on the hardware. I/O handling typically involves reading sensor states into internal memory at the start of a scan and writing output commands to actuators at the end. Separating I/O handling into a dedicated program section improves traceability and allows the rest of the logic to operate on stable, scan-consistent values.

**Ladder Diagram (LD)** A graphical IEC 61131-3 programming language derived from the visual notation of relay-based electrical control circuits. Logic is expressed using horizontal rungs between two vertical rails, with contact symbols representing conditions and coil sym-

bols representing outputs. Ladder Diagram remains widely used in the automation industry, particularly in environments where service technicians are expected to read and diagnose the control logic directly on the controller.

**Mock Module:** A software component that substitutes for a real hardware-dependent module during development and testing. A mock module implements the same interface as the real module but replaces physical I/O signals with simulated values that can be driven from a test harness. This allows the program logic to be executed and verified on a development computer without requiring access to the physical machine.

**Process Image:** A memory region maintained by the PLC runtime that holds a snapshot of all physical input and output values captured at a defined point in the scan cycle. By operating on the process image rather than reading hardware registers directly during program execution, the PLC ensures that all logic within a single scan operates on a consistent set of input values, regardless of when within the scan a particular value is accessed.

**Programmable Logic Controller (PLC):** An industrial-grade computing device designed to control machinery and automated processes in manufacturing environments. A PLC reads signals from connected sensors and other input devices, executes a user-defined control program, and drives outputs to actuators, motors and other devices. PLCs are distinguished from general-purpose computers by their deterministic cyclic execution model, tolerance of industrial environmental conditions, and bounded real-time input and output processing.

**Program Organization Unit (POU):** The fundamental building block of software in the IEC 61131-3 programming model. The standard defines three types of POUs: the Program, the Function Block and the Function. Each POU consists of a variable declaration section and a logic implementation section, and may be written in any of the five IEC 61131-3 programming languages. POUs can call other POUs, with rules governing which types may call which, depending on their respective category.

**Recipe (industrial context)** A structured set of process parameters that define the configuration required to produce a specific product variant on a machine. In an automation context, recipes typically include setpoints such as temperatures, speeds, fill volumes and

timing values. The PLC receives or loads the active recipe and applies its parameter values to the relevant control logic for the duration of the production run. The separation of recipe parameters from machine logic is a recognized design principle in industrial automation, reflected in the ISA-88 batch control standard.

**Structured Text (ST)** A high-level, text-based programming language defined in IEC 61131-3 with a syntax derived from Pascal. Structured Text supports standard control flow constructs including IF-THEN-ELSE, CASE, FOR and WHILE statements, and is considered well-suited for implementing complex logic, arithmetic calculations and object-oriented constructs. It is the primary programming language used in the framework developed in this thesis.

**TwinCAT 3** is the integrated development and runtime environment for PLC and motion control programming produced by Beckhoff Automation. TwinCAT 3 is implemented as an extension to Microsoft Visual Studio and executes on standard PC hardware converted to a real-time controller. Among common industrial PLC platforms, TwinCAT 3 provides the most complete support for the object-oriented extensions of IEC 61131-3, including interfaces, inheritance and abstract methods, making it the target platform for the implementation developed in this thesis.

# Bibliography


Antonsen, T. (2020). *PLC Controls with Structured Text (ST), V3: IEC 61131-3 and best practice ST programming*. Books on Demand, 3rd edition.

Beckhoff Automation (2024). TwinCAT 3 Documentation. https://infosys.beckhoff.com. Accessed: May 15, 2026.

Booch, G. (2007). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, Boston, 3rd edition.

Braun, V. and Clarke, V. (2006). Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3(2):77–101.

Cruz Salazar, L. and Vogel-Heuser, B. (2020). Applying core features of the object-oriented programming paradigm by function blocks. In *Service Oriented, Holonic and Multi-agent Manufacturing Systems for Industry of the Future*, pages 271–284. Springer, Cham.

Freeman, S. and Pryce, N. (2009). *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley, Boston.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston.

Hevner, A. (2007). The three cycle view of design science research. *Scandinavian Journal of Information Systems*, 19(2).

Hevner, A., March, S., Park, J., and Ram, S. (2004). Design science in information systems research. *MIS Quarterly*, 28(1):75–105.

IEC (2003). Iec 61131-3: Programmable controllers – part 3: Programming languages.

IEC (2013). Iec 61131-3: Programmable controllers – part 3: Programming languages.

IEC (2016). Iec 60204-1: Safety of machinery – electrical equipment of machines – part 1: General requirements.

IEC (2021). Iec 62061: Safety of machinery – functional safety of safety-related control systems.

IEC (2025). Iec 61131-3: Programmable controllers – part 3: Programming languages.

ISA (2010). Isa-88.01: Batch control part 1: Models and terminology.

John, K.-H. and Tiegelkamp, M. (2010). *IEC 61131-3: Programming Industrial Automation Systems*. Springer, Berlin, 2nd edition.

Kurose, J. and Ross, K. (2012). *Computer Networking: A Top-Down Approach*. Pearson Education Limited, Harlow.

Kvale, S. and Brinkmann, S. (2009). *InterViews: Learning the Craft of Qualitative Research Interviewing*. SAGE Publications, Los Angeles.

Martin, R. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, Upper Saddle River, NJ.

Martin, R. (2014). *Agile Software Development: Principles, Patterns and Practices*. Pearson, Upper Saddle River, NJ.

Object Management Group (2017). Unified modeling language specification. https://www.omg.org/spec/UML/2.5.1/About-UML.

Oracle Corporation (2026). Java se 26 edition specification. https://docs.oracle.com/javase/specs/.

Rockwell Automation (2024). *Studio 5000 View Designer User Manual*. Rockwell Automation. Accessed: May 15, 2026.

SEK Svensk Elstandard (2023). Iec 62682: Management of alarm systems for the process industries.

Siemens AG (2026). *Totally Integrated Automation Portal V21*. Siemens AG. Accessed: May 15, 2026.

Snyder, H. (2019). Literature review as a research methodology: An overview and guidelines. *Journal of Business Research*, 104:333–339.

Thramboulidis, K. and Frey, G. (2011). Towards a model-driven iec 61131-based development process in industrial automation. *Journal of Software Engineering and Applications*, 4(4):217–226.

# APPENDIX A

PLC folder structure