

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

# Development of functional libraries supporting object-oriented PLC programming of industrial workstations

*Pedro Henrique Ferreira dos Santos*

**Masters Dissertation**

Supervisor: Prof. António José Pessoa de Magalhães

**Masters Degree in Mechanical Engineering**
**Automation Branch**

Faculdade de Engenharia da Universidade do Porto
2023/2024

## Abstract

This document discusses the development of a support library for object-oriented programming (OOP) of programmable logic controllers (PLCs). However, since this programming approach is still not a well-established concept, especially in the PLC domain, object-oriented programming is introduced, discussed, and evaluated in advance.

The text starts by reporting how PLCs have evolved, especially along with the IEC 61131-3 standard, and comparing OOP and traditional procedural programming (PP). To better understand how this paradigm shift impacts industrial software applications and how OOP usage in this context compares to information systems applications, a comparison is made between the use of OOP in information technology (IT) and operational technology (OT). A case study is also presented to compare OOP and PP paradigms in a fairly complex virtual PLC-controlled system. Ending the dissertation, the development of a software library supporting the OOP approach for PLCs is presented, and its utilization is exemplified.

The research conducted concludes that OOP, by dividing the system into objects, allows for programming complex systems in a more organized manner, thereby making software easier to interpret, modify, and expand compared to using PP. However, this new type of PLC programming requires object-oriented supporting tools for problem-solving and software modeling, leading to the widespread adoption of UML and SysML. Consequently, going from PP to OOP in PLC applications requires proper education and training in software engineering.

Therefore, OOP is not a master key, as traditional ladder programming and GRAFCET modeling are still the best options for simple, classical, and well-established control scenarios. As such, OOP does not replace traditional PLC programming, but it is an option for complex systems, where flexibility and adaptability are fundamental.

However, this document is solely an introduction to the topic and, as such, there are many possibilities for continuing the work carried out, such as addressing OOP for human-machine interfaces (HMIs), expanding the developed library, and exploring safety and security using OOP concepts.

Keywords: PLC Programming, IEC 61131-3 Standard, Object-oriented programming, PLC software libraries development.

## Resumo

Este documento discute o desenvolvimento de uma biblioteca de suporte à programação orientada a objetos de controladores lógicos programáveis. No entanto, como este paradigma da programação ainda não é um conceito bem estabelecido, especialmente no domínio dos PLCs, antes de abordar o objetivo principal do documento, a programação orientada a objetos é devidamente introduzida, discutida e avaliada.

O texto começa por abordar a evolução dos PLCs em conjunto com a norma IEC 61131-3 e comparar a programação orientada a objetos com a tradicional programação procedural. De forma a compreender melhor como esta mudança de paradigma impacta as aplicações de software industrial e como o uso da programação orientada a objetos neste contexto se compara às aplicações em sistemas de informação, é feita uma comparação entre o uso deste paradigma de programação nas tecnologias de informação e nas tecnologias operacionais. É também apresentado um case study com o objetivo de comparar a programação orientada a objetos e programação procedural num sistema virtual, relativamente complexo, controlado por um PLC. Para concluir a dissertação, é apresentado o desenvolvimento de uma biblioteca de software que suporta a programação orientada a objetos de PLCs, exemplificando a sua utilização.

Com todo o estudo desenvolvido foi possível concluir que a programação orientada a objetos, ao dividir o sistema em objetos, permite programar sistemas complexos de uma forma mais organizada, tornando o software mais fácil de interpretar, modificar e expandir do que utilizando programação procedural. No entanto, este novo tipo de programação de PLCs requer ferramentas de suporte orientadas a objetos para resolução de problemas e modelação de software, levando à adoção generalizada do UML e do SysML. Consequentemente, passar de programação procedural para programação orientada a objetos em aplicações de PLCs requer uma educação e formação adequadas em engenharia de software.

Portanto, a programação orientada a objetos não é uma chave mestra, uma vez que o tradicional ladder e a modelação em GRAFCET continuam a ser as melhores opções para cenários de controlo simples, clássicos e bem estabelecidos. Assim, a programação orientada a objetos não substitui a programação tradicional de PLCs, mas é uma opção para sistemas complexos, onde a flexibilidade e a adaptabilidade são fundamentais.

Contudo, este documento é apenas uma introdução ao tema e, como tal, existem muitas possibilidades para continuar o trabalho realizado, tais como abordar o uso da programação orientada a objetos para interfaces homem-máquina, expandir a biblioteca desenvolvida e explorar em mais detalhe a segurança utilizando conceitos de programação orientada a objetos.

Palavras-chave: Programação de PLCs, Norma IEC 61131-3, Programação orientada a objetos, Desenvolvimento de bibliotecas de software para PLCs.

## Acknowledgements

I would like to thank Professor António José Pessoa de Magalhães for all the guidance and support during the execution of this document and for introducing me to industrial automation.

I would like to thank my father and mother for always being there for me and for providing me with the best conditions throughout my academic journey.

I would also like to thank my sister for her unconditional support and patience.

# Contents


**1 Introduction** . . . . . **1**
1.1 Dissertation goal and major milestones . . . . . 1
1.2 Document organization . . . . . 2

**2 Towards Object-Oriented PLC Programming** . . . . . **3**
2.1 The early years of PLCs . . . . . 3
2.1.1 Main components of the first PLCs . . . . . 3
2.1.2 Software models and programming languages of the first PLCs . . . . . 4
2.1.3 Practical limitations of the first PLCs . . . . . 5
2.2 IEC 61131-3: a new paradigm in PLC Programming . . . . . 5
2.2.1 IEC 61131-3 software model . . . . . 5
2.2.2 Programming languages . . . . . 7
2.2.3 Unsolved issues and practical problems in the early versions of IEC 61131-3 7
2.3 IEC 61131-3 version 3 and Object-Oriented Programming . . . . . 8
2.3.1 The basis of Object-Oriented Programming . . . . . 8
2.3.1.1 Classes and Objects in IEC 61131-3 . . . . . 8
2.3.1.2 Supporting the four pillars of OOP in PLC programming . . . 9
2.3.1.3 Message passing as a central issue in OOP . . . . . 11
2.3.2 From PLC Procedural Programming to Object-Oriented Programming . 11
2.3.3 Tools for practicing and evaluating Object-Oriented PLC Programming 12
2.4 Chapter summary . . . . . 12

**3 Finding guidelines for Object-Oriented PLC Programming** . . . . . **15**
3.1 OOP contrasts and overlaps in Information and Operational Technologies . . . 15
3.1.1 Information Technology Versus Operational Technology . . . . . 15
3.1.2 UML: the Swiss-knife for Information Systems design . . . . . 16
3.1.3 Finding dual use cases in industrial automation and information systems 17
3.1.3.1 Information system example . . . . . 17
3.1.3.2 Automation system example . . . . . 18
3.1.3.3 Critical appraisal . . . . . 19
3.2 Realizing the practical challenges of Object-Oriented Automation Systems Programming . . . . . 20
3.2.1 Dealing with asynchronous programming . . . . . 20
3.2.2 Dealing with non-reentrant code . . . . . 21
3.3 Resources for managing objects interaction . . . . . 22
3.3.1 Handshake synchronisation support . . . . . 22
3.3.2 Queued data storage and retrieval support . . . . . 24
3.4 Object-Oriented PLC Programming in CODESYS . . . . . 26
3.4.1 Example description . . . . . 26
3.4.2 "Automation devices" Classes . . . . . 26
3.4.2.1 The PickPlace Class . . . . . 27
3.4.2.2 The Conveyor Class . . . . . 28
3.4.3 Supporting Interaction among "Automation" Objects . . . . . 29
3.5 Chapter summary . . . . . 30

**4 A Case Study on going from Procedural to Object-Oriented PLC Programming** . . . . . **33**
4.1 Presenting the case study . . . . . 33
4.1.1 Plant description . . . . . 33
4.1.2 Plant detailed operation . . . . . 35

4.1.3 User interface . . . 36
4.2 Original procedural programming solution . . . 37
4.2.1 Procedural programming of complex hierarchical relations . . . 38
4.2.2 Critical appraisal . . . 40
4.3 Designing and programming a Object-Oriented solution . . . 41
4.3.1 Taking OOP as a new mindset . . . 41
4.3.2 Classes representing "Automation Devices" and "Management Resources" 42
4.3.2.1 The VerifierQueue Class . . . 42
4.3.2.2 The PickPlace Class . . . 43
4.3.2.3 The Conveyor Class . . . 43
4.3.2.4 The TurnTable Class . . . 44
4.3.3 Interaction among objects . . . 46
4.3.4 Critical appraisal . . . 48
4.4 Chapter summary . . . 49

**5 Development of a functional library for supporting Object-Oriented PLC Programming . . . 51**
5.1 Concept and purpose . . . 51
5.1.1 Advantages . . . 51
5.1.2 Development aspects . . . 52
5.2 The developed class library . . . 52
5.2.1 The Queue Class . . . 53
5.2.2 The PickPlace Class . . . 53
5.2.3 The Conveyor Class . . . 54
5.2.4 The TurnTable Class . . . 54
5.2.5 The TwoAxisPickPlace Class . . . 55
5.2.6 The BidirectionalConveyor Class . . . 56
5.2.7 The ChainTransfer Class . . . 57
5.3 Class library application . . . 59
5.3.1 The target workstation . . . 59
5.3.2 UML Modelling of the solution . . . 59
5.3.3 PLC programming . . . 60
5.4 Chapter summary . . . 61

**6 Conclusion and Future work . . . 63**
6.1 Future work . . . 64
6.2 Personal assessment . . . 64

**References . . . 65**

## Acronyms

FB - Function Block
GRAFCET - GRAphe Fonctionnel de Commande, Étapes Transitions
HMI - Human Machine Interface
ICS - Industrial control systems
IEC - International Electrotechnical Commission
IE - Information systems
IT - Information technology
OOP - Object-Oriented Programming
OT - Operational technology
PLC - Programmable Logic Controller
PLCOOP - PLC object-oriented programming
POU - Program Organization Unit
PP - Procedural Programming
SFC - Sequential Function Chart
ST - Structured Text
UML - Unified Modeling Language

# List of Figures


2.1 Main elements of a PLC [8] . . . 3
2.2 Comparison between Ladder Logic and Relay Logic [9] . . . 4
2.3 IEC 61131-3 software model [13] . . . 6
3.1 Sequence diagram of the ATM system . . . 17
3.2 Feeding system scenario . . . 18
3.3 Sequence diagram of the feeding system . . . 19
3.4 "Handshake" class diagram . . . 23
3.5 Handshake sequence diagram . . . 23
3.6 "Queue" class diagram . . . 24
3.7 Pick & place system scenario . . . 26
3.8 "Conveyor" class diagram . . . 26
3.9 "PickPlace" class diagram . . . 26
3.10 Pick & Place system project tree . . . 27
3.11 Methods of the "PickPlace" class . . . 28
3.12 Methods of the "Conveyor" class . . . 29
3.13 Sequence diagram of the operation of the system . . . 30
4.1 Original system [27] . . . 34
4.2 Developed system . . . 34
4.3 Systems elements labeled - palletizing zone . . . 34
4.4 Systems elements labeled - routing/enclosed area . . . 34
4.5 Types of boxes . . . 35
4.6 Representation of the system's operation . . . 36
4.7 Electrical HMI panel . . . 36
4.8 Schematic representation of the original solution [27] . . . 37
4.9 Program "A_CONTROL" [27] . . . 38
4.10 Program "B_NORMAL" [27] . . . 39
4.11 Program "C_AUTO_START_STOP" [27] . . . 39
4.12 Program "D_PROCESSO_1_AUTO_START_STOP" [27] . . . 40
4.13 Sequence diagram of the transport by the pick & places . . . 41
4.14 Sequence diagram of the transport by the turntables . . . 41
4.15 "Queue" class diagram . . . 42
4.16 "Queue" class in CODESYS . . . 42
4.17 "PickPlace" class diagram . . . 43
4.18 "PickPlace" class in CODESYS . . . 43
4.19 "Conveyor" class diagram . . . 43
4.20 "Conveyor" class in CODESYS . . . 43
4.21 Turntable scheme . . . 44
4.22 "TurnTable" class diagram . . . 44
4.23 "TurnTable" class in CODESYS . . . 44
4.24 PLC_PRG in CODESYS . . . 46
4.25 System modes in CODESYS . . . 47
4.26 Code associated with each mode . . . 48
4.27 UML Component diagram of the system . . . 48
5.1 "Queue" class diagram . . . 53
5.2 "Queue" class in CODESYS . . . 53
5.3 "PickPlace" class diagram . . . 54
5.4 "PickPlace" class in CODESYS . . . 54
5.5 "Conveyor" class diagram . . . 54
5.6 "Conveyor" class in CODESYS . . . 54

5.7 "TurnTable" class diagram . . . . . 55
5.8 "TurnTable" class in CODESYS . . . . . 55
5.9 Two axis pick & place . . . . . 55
5.10 "TwoAxisPickPlace" class diagram . . . . . 56
5.11 "TwoAxisPickPlace" class in CODESYS . . . . . 56
5.12 "BidirectionalConveyor" class diagram . . . . . 57
5.13 "BidirectionalConveyor" class in CODESYS . . . . . 57
5.14 Chain transfer table . . . . . 58
5.15 "ChainTransfer" class diagram . . . . . 58
5.16 "ChainTransfer" class in CODESYS . . . . . 58
5.17 Sorting by weight workstation . . . . . 59
5.18 Sorting by weight sequence diagram . . . . . 60
5.19 "ScaleConveyor" class diagram . . . . . 61
5.20 "ScaleConveyor" class in CODESYS . . . . . 61

# 1 Introduction

With the evolution of automation technology, industrial control and production systems are becoming increasingly larger, more demanding, and more sophisticated, making their design, performance, and expandability more complex and critical than ever. This means that modern industrial systems are irreversibly and tightly dependent on the crucial support of industrial software, central to which are industrial control applications running on Programmable Logic Controllers (PLCs). As such, the evolution of industrial systems depends much on the features of future PLCs [1].

This challenge drives the PLC industry - joining manufacturers, programmers, and professional associations - to propose increasingly sophisticated industrial computing practices, trends, and standards. At the same time, in a never-ending feedback loop, PLC users – mainly industrial entrepreneurs and managers – face the growing possibilities of automation technology as a stimulus to achieve more ambitious and flexible productivity goals.

Over the past decades, this never-ending "improvement demanding" cycle has significantly impacted PLCs. In fact, PLCs have long ceased to be thought of and treated as logical devices designed to be programmed by electricians. Due to their current complexity, range of applications, and the need for interaction with many heterogeneous information and operational resources, modern PLCs require programming by automation specialists, namely software technicians with a solid understanding of production engineering and computer science.

The early proposal and later revisions of the IEC 61131 standard, namely part 3 [2], have introduced major milestones in the PLC era. A later and significant focus goes to including object-oriented programming (OOP) concepts in the 3rd edition of the standard. Since then, this programming paradigm - already highly successful in past information systems - has gained increasing interest and popularity in PLC programming. Among PLC programmers, a growing conviction exists that on changing from the classical procedural approach to complex industrial problem-solving, while offering a new perspective on automation systems, object-oriented programming seems to emerge as an advantageous and "de facto" PLC programming solution [3].

In the Section of Automation, Instrumentation, and Control (SAIC) of the Department of Mechanical Engineering (DEMec) at FEUP, PLC object-oriented programming (PLCOOP) has been closely followed with interest. Namely, it was previously considered in Eduardo Silva's Masters Dissertation [4], which laid an interesting groundwork but required further experimentation for conclusions and hopefully convincing validation.

The purpose of this document is to continue this study and investigate upon different views and cases three significant issues: to what extent can object-oriented programming overcome the challenges and limitations of traditional procedural programming, what new learning or re-learning is required on moving to PLCOOP, and whether the results seem to meet the "promised" expectations.

## 1.1 Dissertation goal and major milestones

The main goal of this dissertation is to provide resources that facilitate the understanding and practice of object-oriented programming of PLCs, aiming to develop a class library that serves as the centerpiece of the developed work.

However, although several authors are now considering PLCOOP a promising approach, it is not a well-established topic, and, as such, this dissertation also aims to introduce, develop, and thoroughly analyze it. Therefore, in the move to achieve the final goal, the following milestones are established in advance:

- Analyze the main ideas, concepts, and pillars that support object-oriented programming to understand to what extent it solves the problems of procedural programming (PP);
- Explore modeling techniques suitable for object-oriented programming – namely Unified Modelling Language (UML) [5];
- Compare the use of object-oriented programming in automation and information systems to identify their application and use case similarities, thereby determining if automation systems can draw insights from information systems;
- Practice object-oriented programming in the Codesys development environment [6] in conjunction with Factory IO simulation software [7], a simulation software widely used throughout the automation course.;
- Contrast object-oriented programming with procedural programming in a practical example to validate the presented ideas;
- Contribute to the software development community by creating a class library that enables users to control elements available in the Factory IO application.

It is worth saying that the class library to be developed in the context of this dissertation is not intended to be complete - which would be naturally impossible. It is just a resource that comes as the corollary of the research developed around PLCOOP, and that future automation students can analyze, improve, and work on.

## 1.2 Document organization

Chapter 2 begins by discussing the first PLCs, including their composition, programming methods, and limitations. It then addresses the initial versions of the IEC 61131-3 standard and its impact on PLC programming. Object-oriented programming is then introduced, defining the concepts, ideas, and pillars that support it, allowing for comparison with traditional procedural programming.

Chapter 3 compares object-oriented programming in information and operational technologies to determine if industrial system design and programming can follow in the footsteps of information systems programming and thus inherit the proven good results. This chapter also identifies some challenges associated with using OOP in automation systems and presents resources for managing object interactions. Finally, a simple automation system example programmed using OOP in CODESYS illustrates some of the concepts discussed.

Chapter 4 presents a case study involving reprogramming a relatively complex industrial system, transitioning from procedural programming to object-oriented programming, with the goal of confronting these two programming paradigms in order to validate the ideas presented.

Chapter 5 discusses libraries, explaining their concept, purpose, and key considerations in their development. It also presents and discusses the development and usage of the Factory IO and Codesys based class library that defines the major goal of this dissertation.

Lastly, chapter 6 concludes the dissertation by summarizing the most important conclusions, presenting future perspectives, and providing a personal reflection on the experience of developing this document.

# 2 Towards Object-Oriented PLC Programming

PLCs emerged around 1970 and, since then, they have completely revolutionized the entire landscape of industrial automation. Nowadays, PLCs are responsible for controlling processes, automating and monitoring machines and manufacturing systems, and even ensuring safety in industrial environments. Therefore, it is essential to understand their operation and programming to keep up with emerging trends while ensuring the proper functioning and safety of systems.

This chapter addresses some of the history of PLCs, how they work, and how their programming has evolved over time. Starting with the first programming language used, Ladder Logic, moving through procedural programming, and ending with the main focus of this dissertation: object-oriented programming. Throughout the chapter, the evolution of PLC programming is accompanied by improvements and amendments made to the IEC 61131-3 standard over time.

## 2.1 The early years of PLCs

A PLC is an industrial computer that continuously monitors the status of the input devices and makes decisions based on a custom program to control the status of the output devices. The first PLCs were designed and developed by Modicon to replace relay-based control systems used by GM and Landis. These new controllers eliminated the need for reconfiguration and additional hardware for each new logical configuration while drastically increasing control functionality and reducing the space occupied by the control system [8].

### 2.1.1 Main components of the first PLCs

In terms of functionality, a PLC essentially comprises three main components: inputs, outputs, and CPU (central processing unit). Figure 2.1 presents a schematic representation of the main elements of a PLC.

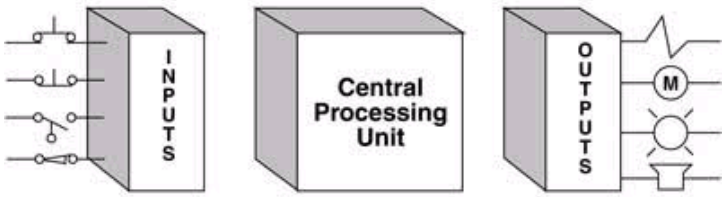



Figure 2.1: Main elements of a PLC [8]

The inputs receive signals from sensors, which gather information about the environment or the system, such as temperature, pressure, liquid level, and presence of objects, among others.

These signals are then processed by the PLC's CPU, which executes the stored program, also known as the control algorithm. This algorithm interprets the input data and determines the output values accordingly.

Finally, the PLC's outputs control the system actuators. Actuators are the devices responsible for executing physical or operational actions based on the decisions made by the PLC. These actions may involve turning motors on or off, opening or closing valves, and triggering alarms, among other operations.

Furthermore, PLCs also feature analog inputs and outputs, communication ports, and high-speed inputs. These components expand the capabilities of PLCs beyond simple digital control, allowing them to interface with analog sensors and actuators, communicate with other devices or systems, and handle high-speed signals for precise control in applications such as motion control or high-speed manufacturing processes.

In terms of operation, a PLC essentially goes through three stages in an endless procedure:

- Input Scan - This is the initial stage and involves reading the inputs by the PLC;
- Program Scan - Knowing the inputs, the PLC runs the program to generate the outputs;
- Output Scan - In this stage, the PLC writes the outputs obtained in the previous stage to its outputs.

There is yet another runtime stage called "housekeeping", which basically involves conducting some internal tests/diagnostics to ensure that the system's optimal performance is maintained.

### 2.1.2 Software models and programming languages of the first PLCs

The PLC's CPU executes the defined program, which implements an algorithm responsible for defining the controlled system operation. This algorithm, which can be developed in various programming languages such as ladder logic, structured text, or function block diagrams, takes the signals received by the inputs to generate the appropriate outputs, thus controlling the system behavior according to the specific needs and conditions of the environment in which it is inserted.

The first programming language PLCs employed was Ladder logic (LD). This is because the first PLCs were developed to replace relay control systems, that is, electromechanical systems. Figure 2.2 illustrates the similarity between Ladder Logic and relay diagrams.

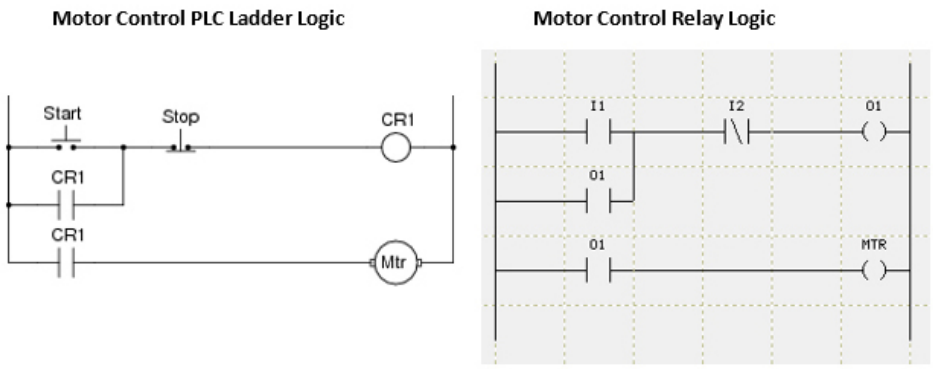



Figure 2.2: Comparison between Ladder Logic and Relay Logic [9]

LD programming language has the advantage of being simple to learn and understand, being familiar due to its similarities with electrical diagrams, being easily debuggable, and representing situations of simple/discrete logic very clearly and objectively.

However, its simplicity also turns out to be one of its biggest flaws. Compared to other programming languages, Ladder Logic tends to fall behind, especially when dealing with large and complex problems and data processing.

Furthermore, Ladder Logic's attempt to mimic the operation of electromechanical systems also presents some issues. One of those issues is that applications that do not operate on electromechanical principles become more difficult to program, as they require significant adaptation to function in Ladder Logic. For example, any operation that involves real values math calculations, such as a PID controller, becomes pretty much impossible for Ladder to handle and because of that new languages, such as Instruction List (IL), started to emerge in order to fulfil what Ladder couldn't.

Therefore, although Ladder Logic maintains its importance in simpler systems, nowadays, due to the increasing complexity and demands of automation systems, the need for more

powerful programming languages is increasing. This leads to the tendency to use high-level languages like Structured Text to handle complex control needs and advanced automation tasks efficiently.

### 2.1.3 Practical limitations of the first PLCs

Despite being a major revolution for the industrial sector, the first PLCs had many practical limitations. Besides their excessive cost and large size, the early PLCs had limited processing power and memory compared to what is available today, which greatly restricted the complexity of the implemented logic and the amount of information/data stored.

Additionally, the first PLCs also had a significant limitation when it came to communication with other devices or networks since, at that time, only proprietary networks were available, posing challenges to their integration into larger systems.

Furthermore, another major problem with the early PLCs was that each PLC had its own programming language and conventions. This made it difficult for engineers to work with different PLCs, as it required them to know all programming languages or just adopt a particular PLC model or brand. Besides that, the fact that each PLC used its own programming language led to non-existent interoperability, as the algorithms developed for a specific PLC were incompatible with another.

Moreover, the fact that PLCs operate by scans causes a fixed delay between the moment a change occurs in an input of the PLC and the moment it detects it. This delay can be a big problem since automation systems are real-time systems, meaning they are "[...] systems in which the correctness depends not only on the logical results of computation but also on the timing of those results[...]" [10], and therefore require timely responses. This made it hard for PLCs to control highly time-critical systems.

Several of these limitations ended up being resolved with the introduction of the IEC 61131-3 standard, which will be discussed in the next section.

## 2.2 IEC 61131-3: a new paradigm in PLC Programming

To address the lack of standardization, the International Electrotechnical Commission (IEC) published in 1993 the first editions of the IEC 61131-3 standard (the third part of the IEC 61131) that specifically deals with PLC programming languages, software and program execution models.

The IEC 61131-3 is "[...] a vendor-neutral and hardware-independent PLC programming standard. The goal of the IEC 61131-3 is to provide uniformity across all compliant PLCs that follow the standard.[...]" [11].

To standardize PLC programming, the IEC 61131-3 introduced new concepts such as "Tasks" and "Program Organization Units" (POUs), along with standardized programming languages. These innovations revolutionized the development and execution of PLC software, boosting flexibility, modularity, and seamless integration with other automation systems and devices. However, the first editions of the standard revealed some practical problems worth noting.

### 2.2.1 IEC 61131-3 software model

The software model for PLCs, as defined in the IEC 61131-3 standard, is structured with a top-level element called a **configuration**. This configuration encompasses the hardware arrangement, memory allocation for I/O channels, and overall system capabilities.

Within this configuration, **resources** are defined to execute programs, akin to processing facilities capable of running IEC programs.

These resources are managed by tasks, which are defined by the IEC 61131-3 as "an execution control element which is capable of invoking, either on a periodic basis or upon the occurrence of the rising edge of a specified Boolean variable, the execution of a set of Program Organization Units (POUs), which can include programs and function blocks whose instances are specified in the declaration of programs" [12]. In other words, tasks are elements that operate either periodically - time-triggered - or triggered by specific events - event-triggered - and control the execution of software blocks within the PLC project, known as POUs.

POUs that compose a program can be reused within the program itself or stored in libraries to be imported and linked into other programs later. Each POU falls into one of the following categories:

- Functions - A function (FC) is a type of POU that contains code and stores data in dynamic memory, meaning that all stored data is lost between scans. Because of that it is used for simple tasks as, for example, adding two numbers. It is not instantiated in the program;
- Function blocks - A function block (FCB) is a type of POU that contains code and stores related data in static memory, meaning that data is preserved from one scan to another. Therefore, this type of POU can be used for more complex tasks such as, for example, controlling a start/stop conveyor belt. It is instantiated in the program and can return more than one output;
- Programs - A program (PRG) is a type of POU that can be put into execution by a task. This type of POUs can be developed using any of the standard programming languages and integrate custom code in the form of functions, or function blocks created within the project or sourced from external libraries. Programs are the only type of POU that can be embedded into a task [4].

Figure 2.3 represents schematically the software model described.

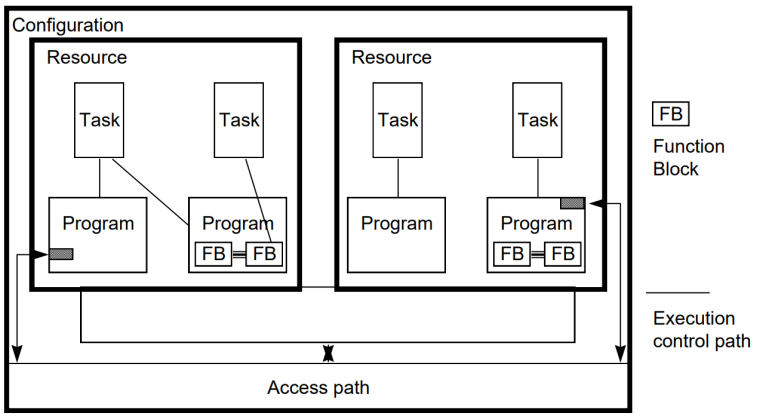



Figure 2.3: IEC 61131-3 software model [13]

In most automation systems, PLCs have multiple programs competing for CPU resources. This concurrency needs to be managed properly by the software application to avoid real-time issues.

Defining the execution mode of a task determines when/how the associated program will be executed. Typically, a task can be executed in three ways: cyclically, in freewheel mode, or by events. A program associated with a cyclically triggered task is executed synchronously at

a specific periodicity. A program associated with a freewheel task is in permanent execution mode, while a program linked to an event-triggered task is executed when the specified event occurs.

Therefore, it is through the use of tasks that IEC 61131-3 addresses these real-time problems. By employing tasks triggered by events, interruptions can be created so that if something doesn't go according to plan (or on time) or some critical change happens in the middle of a scan, an exception routine is initiated. Additionally, IEC 61131-3 also allows defining priorities for tasks, ensuring that if two tasks are requested simultaneously, the one with higher priority is executed first.

### 2.2.2 Programming languages

The first versions of the IEC 61131-3 standard operated on the foundation of procedural programming principles.

**Procedural Programming (PP)** is a type of programming that follows a step-by-step approach to accomplish a task, each step being a procedure. Procedures, also known as functions or routines, consist of a list of logical instructions that the computer must follow in a predefined order, from the beginning to the end.

Procedural programming addresses strongly structured problems using the "top-down approach". This means that the program starts by breaking the problem into smaller subproblems, or sub-procedures, and these sub-procedures are continuously broken down through a process called "functional decomposition" until the obtained sub-procedure is simple enough to be solved [14]. This type of programming is used due to its simplicity, as it is easy to understand and write, and because it is ideal for small projects and simple tasks.

Building upon the procedural programming principles and to achieve uniformity, the IEC 61131-3, defines a set of five standard programming languages: Ladder Diagram (LD), Function Block Diagram (FBD), Structured Text (ST), Instruction List (IL) and Sequential Function Chart (SFC). These languages can be used together or separately, allowing users to develop any control algorithm.

### 2.2.3 Unsolved issues and practical problems in the early versions of IEC 61131-3

In the early versions of IEC 61131-3, practical problems emerged, particularly regarding resource allocation and meeting expectations for PLC utilization. These issues were largely attributed to the inherent limitations of procedural programming.

Procedural programming, while effective for simple tasks, presents difficulties in modeling large and complex systems because, due to the sequential nature of procedural code, it becomes challenging to understand and, therefore, difficult to modify. Thus, this type of programming is not ideal for modeling many real-world situations due to their complexity.

Besides that, the fact that complex systems end up having a multitude of interacting factors makes them somewhat unpredictable, which can really complicate their programming in a sequential manner.

Furthermore, the inability to reuse code across projects, for instance, through the creation of libraries, also leads to the preference for other programming paradigms that offer more flexibility and efficiency in code management and scalability [14].

As such, the shortcomings of procedural programming within the context of the early IEC 61131-3 standards prompted a shift towards alternative programming paradigms, particularly object-oriented approaches, which offer enhanced modularity, reusability, and scalability.

## 2.3 IEC 61131-3 version 3 and Object-Oriented Programming

Throughout the years, changes to IEC 61131-3 were made through two main revisions resulting in new editions of the standard. The third and latest edition of IEC 61131-3 [2], published in 2013, introduced new concepts, such as **classes** and **methods**, that allow the implementation of object-oriented programming [4]. Therefore, by embracing the use of OOP, the IEC 61131-3 aims to tackle the issues associated with procedural programming, promoting modularity, reusability, scalability, and flexibility for a more versatile and structured approach to software development.

### 2.3.1 The basis of Object-Oriented Programming

Object-oriented programming (OOP) is a programming paradigm based on the concept of **classes** transposed to software **objects**, and, more importantly, the **interaction** among them. This type of programming aims to structure/divide the problem into small reusable intervening part types, the classes, which are used to create individual instances of them: the objects. Each class contains data in the form of properties (or attributes) and code in the form of methods (or procedures) [14].

In OOP, each object encapsulates code and data and, almost like an individual program, communicates with other objects to make mutual decisions. Through the methods of a particular object, other objects or the user themselves can interact with it, while its properties serve as indicators of its current state, facilitating accurate interpretation and informed decision-making by other objects.

Therefore, an OOP program ends up resembling a living being in which every cell is an object that communicates and cooperates with others in order to accomplish the primary goals of the program and sustain its functionality.

#### 2.3.1.1 Classes and Objects in IEC 61131-3

As mentioned, object-oriented programming is based on classes and objects. Additionally, each class is associated with methods and properties. In this section, these concepts will be explained.

**Classes**, according to the IEC 61131-3 standard, are program organization units consisting of the definition of a data structure and a set of methods (like subroutines) to be performed upon the data structure [15]. They can be seen/interpreted as the blueprint for creating objects.

**Objects** are instances of a class and can correspond to real-world objects or abstract entities that take part in the program [16].

**Methods**, according to the IEC 61131-3 standard, are language elements similar to a function that can only be defined in the scope of a function block or class type and with implicit access to instance variables of the function block instance or class instance [15]. In other words, methods are functions that are defined within a class to describe the behaviors of its objects.

**Attributes/Properties** are defined in the class and represent the state of an object. Each object of a certain class will have data/information associated with each attribute/property defined in its class [16]. Although not explicitly mentioned in the IEC 61131-3 standard, they can be viewed and implemented as "special methods".

To exemplify the presented concepts, two possible classes will be described.

The first example to consider is a "Conveyor" class, responsible for controlling, as the name suggests, conveyors. As such, it is expected to include methods like "Enable()", to turn on the conveyor, "Stop()", to stop the conveyor, and "ItemCounter()" to update the number of items

on the conveyor by tracking entrances and exits, as well as properties like "pEnableFlag" to identify the state of the conveyor and "pNumItems" to keep track of the number of items on the conveyor at any given moment.

The second example to consider is a "PickPlace" class, responsible for controlling, as the name suggests, pick & place devices. As such, it is expected to include methods like "Move(X,Y,Z)", responsible for moving the pick & place head, "Pick()", responsible for performing the pick operation, and "Place()", responsible for performing the place operation, as well as properties such as "pHasItem", which identifies whether the pick & place is holding an item or not, in addition to the generic methods "Enable()" and "Stop()" and the property "pEnableFlag" already described in the previous example.

These examples will also be used to exemplify other OOP concepts in the next section.

#### 2.3.1.2 Supporting the four pillars of OOP in PLC programming

Literature on OOP mentions four pillars upon which this type of programming is based, ensuring the characteristics that distinguish it from PP. These are: inheritance, encapsulation, abstraction and polymorphism. In order to address these pillars, the IEC 61131-3 incorporates the essential features required to handle them effectively.

This section surveys these pillars, explaining their importance using simple examples.

**Encapsulation**

Encapsulation is a fundamental practice in OOP that involves grouping data with the methods that operate on it within a class, making each class operate almost independently. Furthermore, it allows to restrict access of unauthorized parties to the methods/properties of the class.

To better explain encapsulation the "PickPlace" class presented in section 2.3.1.1 can be considered. The methods that allow controlling the pick & place and the properties that identify its state are defined within the class itself, which means that by instantiating the class, these are automatically defined. Additionally, it is also possible to define which methods/properties can be accessed from outside the class defining which software objects can control the pick & place.

Therefore, in addition to reducing code complexity and promoting reusability, encapsulation enhances security by ensuring that only specific properties and methods can be accessed from outside the class. By deciding which properties/methods to make accessible, the programmer can determine which data can be accessed/altered by third parties.

**Abstraction**

Abstraction is an extension of encapsulation that utilizes classes and objects, which contain data and code, to hide program details from its users. This is achieved through a layer of abstraction between the user and the code. Therefore, objects only reveal the internal mechanisms that are relevant for use by other objects or by the user themselves, concealing any unnecessary implementation code from them.

Considering the "PickPlace" example presented in section 2.3.1.1, abstraction can be easily explained: when calling the "Pick()" method (or any other method from the "PickPlace" class), the user doesn't need to know how it works internally, only that it is responsible for the pick operation. This allows for simplified interaction, focused solely on the desired action and its outcomes. Something very similar happens when controlling the pick & place using an HMI: the operator only needs to know what button to press, even if they don't understand its internal functioning.

Another way to implement abstraction is through **interfaces**. Interfaces function like abstract classes, meaning they are classes whose objective is to express the intent rather than the actual implementation. With an interface defined, classes can be created by implementing it, which requires each class to include all the methods and properties specified in the interface.

For example, an interface "iManipulator" that contains the methods "Enable()", "Stop()", "Move(X,Y,Z)", "Pick()", "Place()" and the properties "pEnableFlag" and "pHasItem" can be used to create manipulator classes. For instance, the classes "PickPlace" (like the one defined in section 2.3.1.1) and "CartesianRobot" can be created by implementing the "iManipulator" interface, thereby including all the specified methods and properties.

**Inheritance**

Inheritance allows classes to inherit characteristics from other classes, enabling a class, the child class, to inherit the properties and methods of another class, the parent class. This enables methods and properties defined in a particular class (the parent class) to be reused, with new classes (child classes) being created by extending the functionalities of the parent class and adding new methods and properties specific to that child class.

For example, considering the class "Conveyor" introduced in section 2.3.1.1, a new "BidirectionalConveyor" class can be created using inheritance. By inheriting the methods and properties of the "Conveyor" class, it is possible to define the normal operation of the conveyor. To make it bidirectional, a new method, "EnableBackwards()", can be added to the new class, enabling it to operate in both directions. This way, the methods originally defined in the "Conveyor" class are inherited by the "BidirectionalConveyor" class, defining its normal operation, and adding the "EnableBackwards()" method extends its functionality to operate in the opposite direction.

**Polymorphism**

Polymorphism refers to the ability of different classes to call a method with the same name, and that method performing differently according to the class that invokes it. There are two types of polymorphism: "compile-time polymorphism", which occurs when the type of the object calling a particular method is defined by the compiler, and "run-time polymorphism" which occurs when the type of the object calling a particular method is defined during the program's execution (run-time). This second type of polymorphism allows a single variable to access different objects while the program is running [17].

Polymorphism often arises in association with inheritance, as a method originally defined in the parent class may need to function differently in the child class. For example, the class "BidirectionalConveyor" used to illustrate inheritance, like the parent class, "Conveyor", has the method "ItemCounter()". However, considering that in the case of the bidirectional conveyor there are two possible directions for item transit, it is necessary to redefine this method. Thus, both the "Conveyor" class and the "BidirectionalConveyor" class contain the "ItemCounter()" method, but its operation depends on the class.

Polymorphism is also very commonly associated with interfaces, as the various classes implementing an interface can have different ways of executing the same method. For example, the classes "PickPlace" and "CartesianRobot" used to illustrate abstraction through interfaces have the same methods since both implement the "iManipulator" interface. However, each method can be defined differently for each class because, although both classes represent manipulators, the way the pick & place performs a specific operation is completely different from how the Cartesian robot does.

#### 2.3.1.3 Message passing as a central issue in OOP

All the concepts around OOP presented so far, such as classes, objects, and inheritance, are undoubtedly essential to grasp this type of programming. However, there is one concept that has not yet been addressed and that is the most important of all: messaging.

This concept is regarded as the most important by Alan Kay, who was one of the inventors of the programming language "SmallTalk", one of the founders of the concept of object-oriented programming, and the one who coined the term OOP. According to Alan Kay, the focus of object-oriented programming is not on classes or objects but rather on the communication between them: "[...] I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea. The big idea is "messaging" [...] " [18].

It is the communication among the various objects of a program that allows it to function naturally, almost like a living organism. This happens because each system element operates independently, receiving and sending information to others. Therefore, changes in individual elements (when correctly applied) do not disrupt the functioning of the system as a whole.

From here, the significant difference between PP and OOP arises. Programs in PP operate like mechanical clocks, where all the gears must mesh perfectly for the program to function correctly, any small issue in one of the gears leads to program failure. This methodology results in lengthy programs that are rigid and difficult to modify or expand. On the other hand, OOP operates more like a living organism composed of cells, the objects, where each cell is quite simple and executes its function individually, capable of undergoing changes without disturbing the organism as a whole. This methodology results in flexible programs that are adaptable to user needs and easy to alter/expand [19].

### 2.3.2 From PLC Procedural Programming to Object-Oriented Programming

By embracing Object-Oriented Programming, the IEC 61131-3 sets out to tackle various issues encountered in Procedural Programming. This proactive approach reflects a commitment to improving code structure, fostering code reuse, and simplifying software development processes in industrial automation.

One of the main advantages of object-oriented programming is that it promotes **modularity** through encapsulation, by modeling systems through small and simple structures, the objects. This makes the program much more organized than procedural code which makes it easier to develop and debug.

By using classes, OOP allows to drastically reduce the amount of code needed, as a single class can create countless objects. This allows for **easy expansion/alteration** of the code since objects can easily be added/removed to the existing program.

Besides that, the fact that **properties and methods can be inherited** from other classes makes it easier to create new classes since they can simply use the properties and methods already defined in the existing ones. This also reduces the amount of code needed by **reusing code**, which makes it easier to expand the existing program.

Through polymorphism, OOP enables a piece of code to exist in multiple versions, adjusting its behavior according to different settings or environments. This makes the code more **flexible**, which allows it to better adapt to possible changes.

Another major feature of OOP is that it allows programmers to create programs from pre-written code instead of having to start each project from scratch. From that, libraries can be used to reutilize code that was developed for other projects. This makes tackling large projects more manageable by leveraging existing code, reducing development time, and enhancing overall code reliability by reusing of tested and proven code.

Last but not least, object-oriented programming also **enhances security** and code readability through encapsulation and abstraction by concealing unnecessary information from the user.

Therefore, object-oriented programming emerges as a superior solution for developing complex systems and large-scale projects. PP's limitations, as detailed in section 2.2.3, underscore the necessity for a more organized, adaptable and efficient approach, which OOP aptly provides. By promoting modularity, code reuse, and easy expansion, OOP fosters efficiency and scalability in software development. Moreover, its emphasis on encapsulation enhances code reliability and security while improving flexibility through polymorphism.

### 2.3.3 Tools for practicing and evaluating Object-Oriented PLC Programming

With the evolution of automation systems, the limitations of procedural programming become increasingly notorious. Procedural programming, by its nature, is not suitable for developing complex systems composed of multiple elements like automation systems. To achieve this, PP would have to be able to better organize the code in order to facilitate its modification and expansion, as well as to enable code reuse between programs.

In this context, object-oriented programming emerges as a compelling solution. By utilizing objects, OOP divides large programs into smaller, more manageable parts, each encapsulating a set of methods and properties necessary to control specific system elements. This modular approach allows each object to function independently, facilitating seamless code reuse between programs and significantly easing expansion and modification efforts.

In order to leverage object-oriented programming effectively, it is essential to utilize a compatible development environment. Nowadays, there are several alternatives, with one of the most commonly used being CODESYS.

CODESYS is a development environment developed and distributed by the German software company 3S-Smart Software Solutions for programming controller applications in accordance with the international standard IEC 61131-3 [6]. CODESYS supports the utilization of function blocks, methods, and interfaces as outlined in the IEC 61131-3. However, it does not incorporate support for the class concepts. Instead of that, CODESYS allows the creation of function blocks to which can be added methods and properties in order to use them as classes.

One of the main advantages of CODESYS is the fact that many PLC manufacturers, including Schneider Electric and Beckhoff, incorporate the CODESYS framework into their application design software. This means that knowing how to program in CODESYS equips engineers with the skills needed to work with Beckhoff and Schneider PLC platforms.

Throughout this dissertation CODESYS will be the software used to develop the presented programs.

## 2.4 Chapter summary

In this chapter, the history of PLCs, their operation, and initial programming, with Ladder Logic as the predominant language, were addressed, along with a discussion of some of their initial limitations.

With the increasing complexity of the systems to be controlled and the introduction of the IEC 61131-3 standard, Ladder Logic began to lose ground to other procedural programming languages which offer greater capabilities for programming complex systems. However, the lack of code reusability and its sequential nature lead to complex, lengthy, and repetitive code. Consequently, procedural programming is not the ideal choice for complex systems, being more suitable for simpler situations.

Therefore, object-oriented programming emerges as a solution. Despite requiring much deeper knowledge from programmers, object-oriented programs end up functioning almost like a living being, where each object operates independently, communicating and cooperating with others to collectively achieve the user's intended goal, allowing for the control of large systems while minimizing the amount of necessary code, ensuring greater security, and providing ease for future system alterations and expansions.

The next chapter will address a more practical analysis of OOP, comparing its use in automation with other areas, presenting tools for OOP system modeling, and providing a conceptual example demonstrating the implementation of OOP in CODESYS.

# 3 Finding guidelines for Object-Oriented PLC Programming

As stated in the previous chapter, object-oriented programming revolutionizes complex software development by allowing the creation of large systems through the composition of small, self-contained units. By breaking down systems into manageable components, OOP promotes modularity and flexibility, allowing code reuse and facilitating software changes. This leads to more efficient and scalable programmable solutions.

However, object-oriented design and programming did not come to life in industrial computing but in information systems, where it led to some paradigm shifts, starting with the modeling/specification itself, as traditional flowcharts became insufficient. This led to the adoption of much more comprehensive and versatile designing tools, particularly "Unified Modeling Language", commonly known as UML [20].

The objective of this chapter is to understand whether a similar leap should be made in industrial computing, evolving, for example, from GRAFCET to UML [5] or SysML [21], and furthermore, to understand to what extent industrial systems design and programming can follow in the footsteps of information systems programming and thus inherit the proven good results obtained there.

Therefore, in this chapter, a pragmatic analysis of object-oriented design and programming, in the context of the industrial applications, is conducted. This analysis compares the application of object-oriented principles in operational technology to their application in information technology, with emphasis on one of the most widely used tools for modeling OOP systems: UML. Additionally, some of the challenges associated with utilizing OOP for automation systems are addressed. Finally, to show the implementation of some of the concepts covered in CODESYS, a conceptual example of an industrial system is presented.

## 3.1 OOP contrasts and overlaps in Information and Operational Technologies

Whilst this document is centered on the application of OOP in operational technology (OT), particularly in PLC programming, it is important to remember that object-oriented programming was originally used in information technologies (IT), for example, in business-related information systems (IS), and only recently has it begun to be used for operational technology such as automation systems.

Therefore, a comparison between the usage of OOP in information technology and operational technology, outlining the similarities and differences between these areas, is a natural starting point in order to understand how such different systems can be programmed in such a similar way.

### 3.1.1 Information Technology Versus Operational Technology

**Information technology** focuses on the processing and transmission of computer data, encompassing networks, servers, databases, and software utilized for the efficient handling and utilization of information within businesses [22]. These technologies improve whenever data storing, transmitting and processing capabilities becomes faster, as their performance is measured in services per second.

IT is the core of **information systems** which are interconnected set of components used to collect, store, process, and transmit data and information. It comprises hardware, software, data, people, and processes that work together to transform raw data into useful information [23].

Most modern companies and organizations rely entirely on information systems to conduct and manage their business operations, dynamically interact with their customers and suppliers, and maintain their competitive position in the market. Additionally, information systems play a crucial role in managing interorganizational supply chains, facilitating the flow of products and services, as well as in enabling electronic markets, where commercial transactions are conducted swiftly and securely [23]. However, product flow, process, and storage predominantly rely on operational technology rather than information technology.

**Operational technology** refers to the hardware and software systems that are used to control and monitor physical processes. Essentially, OT encompasses the devices and applications that directly impact the operational efficiency of industrial equipment, factories, plants, and other physical infrastructures, as such, it is designed for real-time interactions with the physical world [22]. So, its performance is measured in control tasks per CPU, therefore, OT does not necessarily improve with faster computer technology, as control tasks "just" require "fast enough" running software, i.e., compatible with the timing constraints of the controlled systems.

OT is applied in **industrial control systems (ICS)**, such as automation systems, which encompass various control systems and their accompanying instrumentation, comprising devices, networks, and controls utilized for the operation and/or automation of industrial processes [24].

In summary, information technology focuses on the processing and transmission of data, enabling efficient handling and usage of information. It is the core of information systems that collect, store, process, transmit and transform data into useful information being crucial for modern companies' operations. On the other hand, operational technology encompasses hardware and software systems used to control and monitor physical processes, particularly in industrial plants, and automation systems. Additionally, IT improves as technology becomes faster, however, the same does not happen with OT because, unlike IT which aims to deliver its services as quickly as possible, OT does not rely as much on speed but much more on processing power.

### 3.1.2 UML: the Swiss-knife for Information Systems design

The adoption of OOP in IS has instigated a notable paradigm shift, particularly in systems modeling. Traditional flowcharts now fall short of capturing the essence of OOP, since, unlike strictly procedural programming, it does not operate sequentially. From here, it emerges the need for a new modeling tool that focus on representing interactions between objects rather than the sequence of actions, leading to the adoption of UML.

UML is a standardized modeling language that can be used by all object-oriented systems. It is composed by a set of diagrams, such as class diagrams, sequence diagrams and use case diagrams, that assist developers in specifying, visualizing, constructing, and documenting their systems and software. UML represents a set of best engineering practices that have proven effective in modeling large and complex systems. As such, it is an essential tool that enables development teams to communicate, discuss, and explore new designs for their systems [5].

As automation systems adopt OOP, the same issue arises, as traditional GRAFCET becomes insufficient to model object-oriented systems since they do not function sequentially. Therefore, also arises in automation systems the need for a modeling tool capable of representing interactions between system objects, similar to how UML is utilized in information systems, consequently leading to the adoption of SysML.

SysML, "Systems Modeling Language", is a modeling language based on UML that extends and adapts it to better align with the requirements of modern systems engineering. Just

like UML, SysML operates as both a graphical and textual language, facilitating effective communication and documentation of system designs. As it is based on the syntax as UML, it uses similar concepts and rules for defining and relating its elements.

However, SysML diverges from UML in certain aspects, reflecting its broader scope and objectives within systems engineering. Unlike UML, SysML encompasses a wider range of systems engineering disciplines and considerations, extending beyond the confines of software engineering. To meet these broader requirements, SysML has simplified and adapted certain concepts and diagrams, tailoring them to the nuances of systems engineering [21].

Therefore, while SysML is arguably more suitable for system control, its overlap with UML is the most relevant part for this document, consequently, preference will be given to UML. In the scope of this document, only the essentials of UML class diagrams and sequence diagrams will be used to illustrate ideas and developments, thereby leaving out many of UML's tools and features.

### 3.1.3 Finding dual use cases in industrial automation and information systems

In order to understand if it is possible to follow in the footsteps of information systems regarding the use of OOP by automation systems, it is necessary to first understand if these two types of systems are similar in the way they rely on OOP. Therefore, in this section, two examples are presented, one from an automation system and another from an information system, which are then compared to understand their OOP dualities.

#### 3.1.3.1 Information system example

In this example, the most common application presented in books about information systems is used: an automated teller machine (ATM). This system allows replacing a bank employee, enabling the user to, for example, make deposits or withdrawals without having to go to a bank.

To better understand how this system works, the following use case scenario will be considered: a user inserts their credit card into the ATM, which prompts them for their PIN. If the inserted PIN is correct, the user is then presented with the set of available functionalities. Selecting the withdrawal option, and after verifying the account balance, the machine dispenses money to the user. Figure 3.1 shows the sequence diagram for the described scenario.

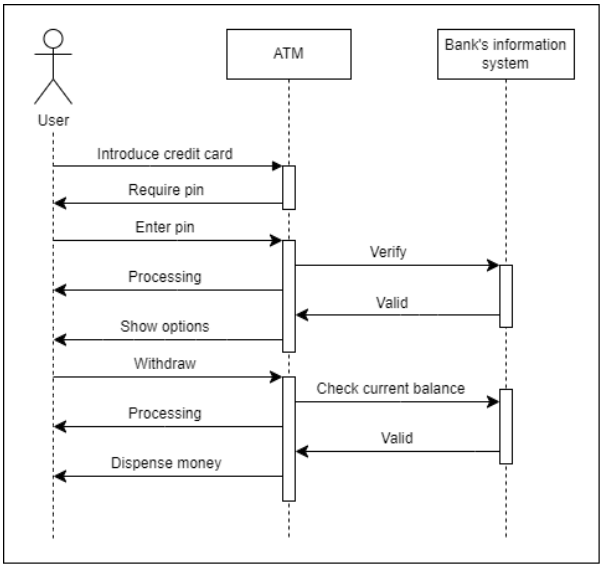



Figure 3.1: Sequence diagram of the ATM system

The system consists of an agent (the ATM user) and two main objects: the ATM terminal and the bank's information system. The ATM serves as an interface with the user, allowing services request, while communicating with the bank's information system to confirm that the requested operation is authorized.

This example is a typical use case where object-oriented programming is the ideal approach since the system does not function sequentially. For instance, when the user enters the PIN, card information is sent to the bank for validation. Depending on the bank's response, the sequence of actions within the ATM itself changes. The same applies to when the user tries to withdraw money: before dispensing it, the ATM "asks" the bank if the account in question has enough balance. Only in this case is the money handed over to the user. Otherwise, the ATM informs the user that the requested operation is not possible.

OOP becomes the preferred programming paradigm to apply in such cases, as the systems don't work sequentially but rather through the interaction of different elements, leading to varying courses of action depending on the circumstances. In these cases, using procedural programming would be considerably more complex and less flexible.

#### 3.1.3.2 Automation system example

Many ATM "equivalent" use cases can be found in automation systems. For example, the box feeding system represented in the Figure 3.2.

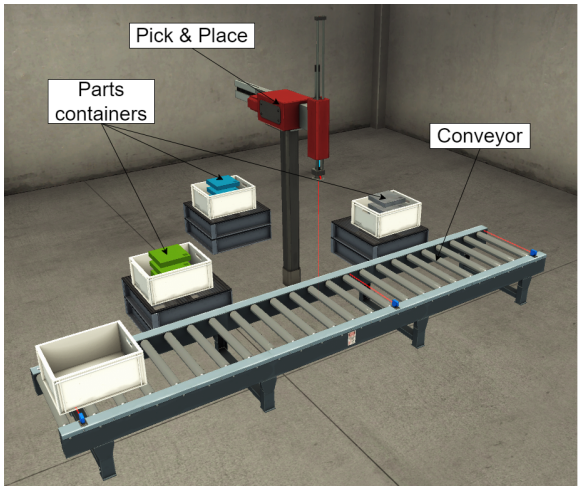



Figure 3.2: Feeding system scenario

This system comprises a conveyor, a pick & place, and three parts containers. Its mission is providing each box that enters the conveyor with the set of parts it requests. To do so, each box entering the conveyor has an RFID tag containing the list of parts to be loaded. When the box enters the pick & place's range, the conveyor stops, and the pick & place reads the tag, getting information about the parts needed to supply the box. Once the order is fulfilled, the conveyor resumes its movement.

However, considering that there may not be enough parts available in the containers, before starting the service, the pick & place queries a SCADA database to ensure that it will be able to fulfill the order. If the requested parts are not available, depending on whether the "customer" box has low or high urgency (which can be defined in the RFID tag), either the service is not initiated, being the empty box transported by the conveyor to await restocking, or the service is carried out but only partially fulfilling the order.

Figure 3.3 shows the sequence diagram that represents a situation where a high-urgency box is placed on the conveyor and there are not enough parts to complete its order.

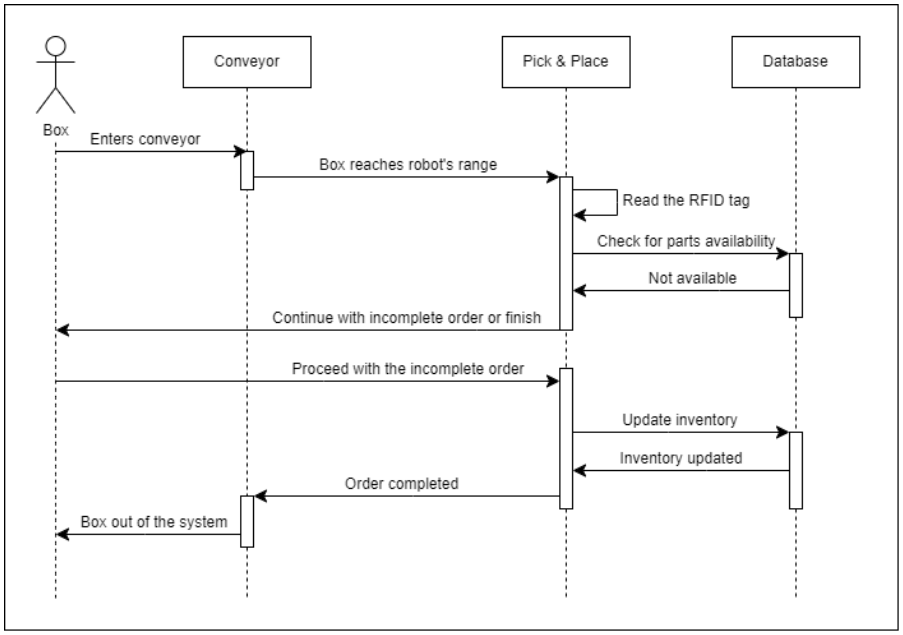



Figure 3.3: Sequence diagram of the feeding system

In this case, the system is divided into three objects: the pick & place (that encapsulates an RFID reader), which directly interacts with the user (boxes to be fed with parts); the SCADA database, which enables the pick & place to know if there are enough parts to fulfill the order; and the conveyor, which facilitates the transportation of the boxes.

Like in the previous example, the system doesn't function sequentially but rather through interactions among its elements. Depending on the "response" from the database to the pick & place's "query", the sequence of actions for the same box can be completely different. Therefore, as in the previous example, object-oriented programming emerges as the ideal solution over procedural programming.

#### 3.1.3.3 Critical appraisal

Analyzing the two examples, and starting by the similarities, one realizes that despite information and automation systems having completely different objectives, they both can employ object-oriented programming very effectively since, as observed, both systems operate in a non-sequential manner (especially in relatively more complex cases), which highlights the superiority of OOP over procedural programming in such contexts.

Another interesting comparison is related to the objects each of the systems utilizes. Despite being associated with completely different elements and having completely distinct functionalities, there is an obvious duality between the objects of the two systems, specifically:

- ATM users in the first example are equivalent to the boxes in the second example, as both access the system with the aim of being served in some way;
- The ATM in the first example is equivalent to the pick & place in the second example, as both are intended to serve the user directly, whether it is a person in the case of the ATM or a box in the case of the feeding system;
- The bank's information system is equivalent to the SCADA database in the feeding system, as both are used to "answer questions" from an actor (the ATM in the first example or the pick & place in the second example) so that it knows how to act.

Therefore, through the comparison of the objects used in both types of systems, it becomes clear that programming for both systems will involve using objects that, although different, are equivalent. This results in programs that, despite having completely different objectives, ultimately approach the problem in quite similar manners.

However, the fact that they deal with completely different realities also results in noticeable differences, such as:

- Unlike people who naturally approach the ATM, the parts of the feeding system need to be transported, hence the necessity of using and controlling a conveyor;
- In contrast to the ATM example where people know the service they need and interact with the ATM to obtain it, in the feeding system example, the boxes can't communicate with the pick & place directly, hence the need for the RFID tag system and a database supporting it.

Therefore, ATM users can be seen as "smart boxes" as they control themselves.

Analyzing and comparing these two examples, it can be concluded that despite information systems and automation systems having completely distinct objectives, both make good use of OOP since, as systems grow in complexity, they inherently transition to non-sequential operations. Consequently, requirements and responsibilities have to be broken down and assigned to different actors that cooperate among themselves in pursuit of a global goal.

This complexity underscores the limitations of procedural programming, which struggles to effectively manage such intricate systems. Besides that, despite having completely distinct objectives, these two types of systems end up using analogous objects, resulting in their programming being quite similar. Thus, it is possible to conclude that as OOP begins to be applied in automation systems, there is much to learn from the way OOP is already utilized in information systems.

## 3.2 Realizing the practical challenges of Object-Oriented Automation Systems Programming

As stated, OOP was initially developed for information systems. Since these systems operate solely on information, differences arise when applying OOP to automation systems driving physical elements. This section addresses some of these differences and outlines related consequences.

### 3.2.1 Dealing with asynchronous programming

One of the main differences between automation systems and information systems is that information systems rely on synchronous programming, while automation systems rely on asynchronous programming. This difference not only impacts, as expected, the development of the systems, but also leads to differences in the use of UML for both types of systems.

**Synchronous programming** means that the code is executed sequentially from start to finish, and a line is only executed when the preceding one has finished [25]. This type of programming is used in situations where a single execution of a procedure (from start to finish) is sufficient to achieve the final goal, which means that the task is fast enough to keep up with the code execution. Otherwise, the processor is monopolized by the procedure for a considerable time while other procedures are postponed.

Synchronous programming is typically found in systems that are data driven and do not depend on the real world interactions through inputs and outputs (like sensors and actuators), such as an algorithm to sort an array or an algorithm to add two numbers. Therefore, this type

of programs follows a clear and predictable flow, as each step depends only on the completion of the previous one.

**Asynchronous programming** allows multiple processes to run independently without significantly blocking each other. In an asynchronous program, instructions don't need to wait procedures to finish before they can be executed [25]. This type of programming is used when a single execution of a procedure (from start to finish) takes too much time, thus, the procedures keeps running outside the code line execution, while its execution can be monitored at each procedure execution call. In other words, asynchronous programming supports the progressive execution of a (real world) task that is much slower than the code execution. Thus, the procedure is limited to periodically monitoring the process, being able, for example, to correct it or abort it, since if the procedure continuously monitored the process, the possibility of concurrency, a fundamental characteristic of controllers, would be lost.

Asynchronous programming is typically found on systems that rely on real-world interactions, such as automation systems using sensors and actuators. This is because asynchronous programming enables code to progress concurrently instead of just "waiting" and "atomically" supporting relevant I/O signals.

Therefore, asynchronous programming makes automation systems opt for communication through shared variables instead of discrete messages as synchronous programming does. This happens because, being asynchronous, often requests from one object cannot be immediately accommodated by another since it may be busy, requiring messages to be constantly resent or placed in a queue. By using shared variables, each object knows the state of others and initiates its activity when it deems appropriate.

This way of communicating between objects ends up complicating the representation of this type of systems through UML sequence diagrams since, as they were developed for synchronous programming, they only support the representation of communications through messages.

Therefore, when using these types of diagrams to model automation systems, the representation obtained is often not entirely accurate, but nonetheless, it still effectively explains its operation.

### 3.2.2 Dealing with non-reentrant code

Another significant distinction between automation systems and information systems programming lies on computer tasks execution. Information systems mostly rely on reentrant programs and functions, whereas automation systems rely on non-reentrant execution. This discrepancy not only influences the development of the system but also leads to differences in how UML is utilized in each case.

**Reentrant functions** allow multiple programs to invoke, interrupt, and restart them simultaneously [26]. In other words, reentrant functions are functions that can have more than one instance of themselves running simultaneously.

This type of function is typically found in IT since, most of the time, functions can be called again during execution, allowing for multiple code and data instances of the same function to be running at the same time. For example, in a bank, almost all functions are reentrant since multiple customers can request the same service simultaneously. However, it is important to note that even in information systems there are exceptions, such as functions that directly access the database, which cannot be reentrant at some level.

**Non-reentrant functions** are the opposite of those described in the previous point, meaning they are functions that once in execution cannot be invoked, interrupted, or restarted by other programs. This type of function is rare in information systems, being more common in physical systems like automation systems.

Typically, the use of non-reentrant functions is associated with some form of processing power limitation within the system due to resources that cannot be shared such as a memory zone in IT or a machine in OT. Therefore, unlike information systems, automation systems primarily operate through non-reentrant functions because they rely on physical elements, which can easily become "busy" and prevent new routines from starting. A possible example would be a simple pick & place operation: while it is transporting one piece, it cannot receive another, rendering the transport function non-reentrant.

In the case of OOP, unlike function blocks, objects have multiple methods, and each of these methods can be either reentrant or non-reentrant. Typically, methods associated with indivisible processes, technically known as atomic processes, are non-reentrant. These methods function as a kind of service counter/"guichet" where the service is requested and eventually placed in a queue or refused depending on the system's state. In IT, transactions with atomic phases exist, but these are very short and have little effect on response time, allowing a server to serve multiple clients simultaneously.

This difference causes automation systems, which predominantly use non-reentrant functions, to be somewhat unpredictable. This happens because, in contrast to information systems, where specific operations consistently follow the same sequence of actions, automation systems introduce variability, with the same operation potentially following different sequences depending on the system's state. For instance, if the system is busy when a certain operation is requested, the sequence of actions might include a waiting period that wouldn't occur if the system were available.

This variability also ends up complicating the modeling of automation systems through sequence diagrams, as it causes there to be several possible sequences and thus several possible sequence diagrams for a given operation depending on the state of the system. Therefore, fully representing an automation system requires many sequence diagrams, yet it is common to use a single diagram as it can capture the core operational flow of the system.

## 3.3 Resources for managing objects interaction

Just like "computing objects", "operational objects" - i.e., those relevant to automation, such as robots, conveyors, and machines - operate in concurrent and collaborative environments and therefore can coexist and interact with one another. However, if they are not properly managed/supervised, they can cause operational conflicts, inefficiencies, or even failures. Therefore, it is important to include "management objects" in the programming to adequately regulate the interaction between these objects, similar to how operating systems manage processes on a computer.

This section addresses this problem by focusing on two management resources that, while not particularly visible or tangible, are crucial in both types of systems: queues and handshakes. These elements are responsible for ensuring proper interaction between the objects that constitute the system, guaranteeing synchronization between the elements and allowing task concurrency.

### 3.3.1 Handshake synchronisation support

The handshake is primarily a **synchronization** protocol between a producer and a consumer. The concept of handshake is closely associated with networks and allows for establishing reliable communication between two devices, ensuring that before initiating communication, both devices are available and ready to exchange information.

Although more commonly associated with communication between PLCs, the concept of handshake is essential for object-oriented programming as it allows for **synchronizing** two objects to ensure that both are ready to perform a specific simultaneous or sequential task.

In OOP it is common to define a handshake as a class. This class can then be instantiated, allowing the definition of various handshakes between the various elements of the system. When it comes to the handshake itself, there are several possible types, with one of the most common in automation systems being the binary handshake. Figure 3.4 diagram models a binary handshake class.

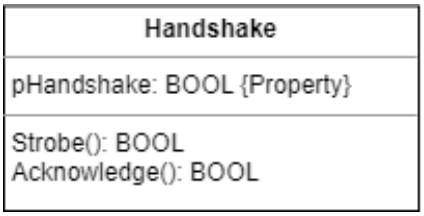



Figure 3.4: "Handshake" class diagram

The "Handshake" class is very simple as it doesn't have any inputs or outputs. Its sole function is to control the handshake between two elements of the system and, to achieve that, only one property and two methods are needed:

- The "pHandshake" property is a simple boolean that can be altered by the two class methods;
- The "Strobe" method takes no inputs and is used by the producer to set the "pHandshake" property to TRUE in order to signal to the consumer that there is information to be received;
- The "Acknowledge" method takes no inputs and is used by the consumer to set the "pHandshake" property to FALSE in order to signal to the producer that the information has been received.

To make this work, the handshake needs to function as a shared variable so that both the producer and consumer have access to its methods and properties. In CODESYS, this can be achieved by defining the handshake as "VAR_IN_OUT" variable within the classes of both the producer and consumer, allowing them to access the methods "Acknowledge()" and "Strobe()", as well as the property "pHandshake".

Figure 3.5 shows a sequence diagram that represents the operation of a binary handshake.

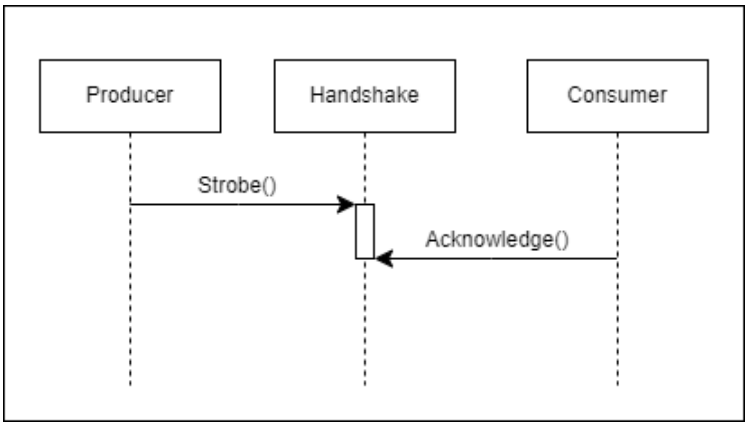



Figure 3.5: Handshake sequence diagram

The producer checks the "pHandshake" property waiting for it to become FALSE, identifying that it can send new information. When the variable is FALSE, the producer sets it to TRUE by invoking the "Strobe" method when it wants to send information. The consumer also constantly checks the "pHandshake" property, waiting for it to become TRUE, identifying that there is information to receive. When the variable is TRUE, and after obtaining the information or completing its task, the consumer sets the variable back to FALSE by invoking the "Acknowledge" method, and then waits for a new handshake.

It is also important to note that the sequence diagram in Figure 3.5 does not accurately represent the operation of the handshake since it is time-triggered, meaning the communication between the elements is asynchronous and, as mentioned in section 3.2.1, UML sequence diagrams are not ideal for representing this type of communication. Nevertheless, it does allow for understanding the basic operation of the handshake.

Another way to use handshakes would be to define the "pHandshake" property within the producer (or the consumer) itself. However, to maintain program organization, and to have each object responsible for a single task, it is preferable to use the "Handshake" class.

Therefore, by using handshakes, it is possible to ensure that a particular operation only occurs when the objects involved in it are synchronized. In other words, before an object initiates a task involving other objects, it must first, through handshakes, ensure that the other intervening objects are available to perform it.

### 3.3.2 Queued data storage and retrieval support

A queue is a very common buffering element in both information and automation systems that allows organizing/managing the flow of items awaiting processing. One can think of queues in a supermarket, a bank, or automation situations like a queue of parts waiting to reach a pick & place.

These elements are essential in object-oriented systems because, due to the limitations of the system itself, whether they are processing limitations in information systems or physical limitations in automation systems, there is a maximum number of services that can be performed simultaneously. This translates into a maximum number of clients that can be served simultaneously by an information system (such as in a bank) or into a maximum number of operations that can run simultaneously in an automation system (such as transport operations in a pick & place system). As such, and to maintain **concurrency**, a key characteristic of these systems, it is essential to ensure that if a request cannot be fulfilled immediately, it is placed in a queue where it waits until the system becomes available to serve it.

For example, in the case of a bank, numerous requests arise simultaneously from diverse sources like ATMs or the bank's online platform. Therefore, these requests must be placed in a queue since the bank's information system cannot respond to them all at once. Similarly, in an pick & place system, the boxes on a conveyor are placed in a queue, waiting for the pick & place to become available since it can only transport one box at the time.

Figure 3.6 represents an example of a "Queue" class.

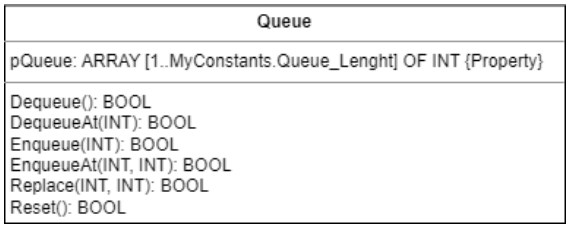



Figure 3.6: "Queue" class diagram

In this example, the "Queue" class has neither inputs nor outputs, as its sole and exclusive function is to control the queue. The queue itself is an internal variable within the class, structured as an array of integers and associated with the property "pQueue". Therefore, it can be accessed by external elements through the property.

The methods used in this example are described as follows:

- The "Enqueue" method takes an integer as input and is responsible for inserting the input element into the first available empty slot of the queue;
- The "EnqueueAt" method takes two integer inputs: the value to be added and the position where it should be inserted. Its function is to insert the input value at the indicated position in the queue, ensuring that all preceding elements shift backward by one unit;
- The "Dequeue" method requires no inputs, and its function is simply to delete the first element of the queue and advance the remaining elements by one position;
- The "DequeueAt" method takes an integer as input, and its function is to remove the element from the specified position and shift the preceding elements forward by one unit;
- The "Replace" method takes two integer inputs: the value to be added and the position where it should be inserted. Its function is to replace the element at the specified position with the input value;
- The "Reset" method requires no inputs, and its function is simply to clear the queue by deleting all of its elements.

Additionally, being programmed through OOP allows complexity to be significantly increased without much difficulty. For example, in this case, instead of using an array of integers for the queue, an array of arrays could be used. In other words, each element of the queue would be a structure/array containing various information about the respective element. Despite this change seeming complex, thanks to OOP, it would be quite simple as it would only require changing the array type.

Thus, the presented example is a good basis for a queue that can be easily added to any system and readily expanded by adding new methods and properties without needing to modify the existing structure.

## 3.4 Object-Oriented PLC Programming in CODESYS

In this section, a practical example illustrating the use of object-oriented supporting an automation task will be presented. The goal is to demonstrate the implementation in CODESYS of some of the concepts about OOP that have been addressed throughout the document, while also serving as a useful example for understanding their application.

### 3.4.1 Example description

Figure 3.7 presents the system to consider.

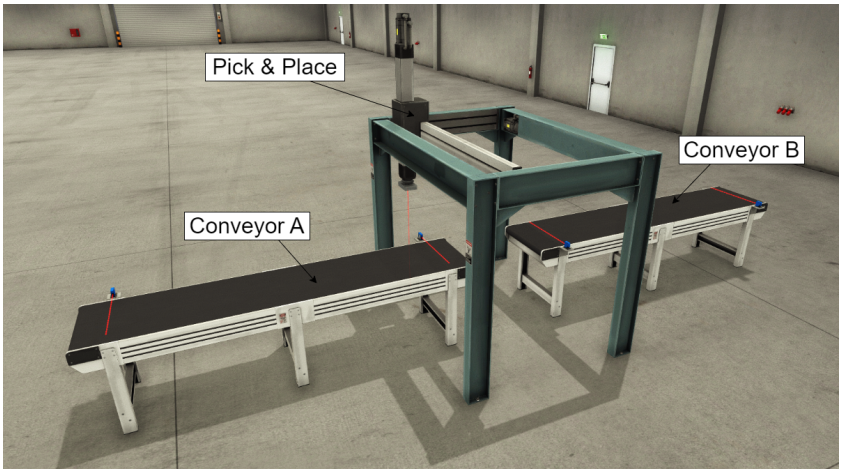



Figure 3.7: Pick & place system scenario

It consists of a pick & place and two conveyors: conveyor A and conveyor B. Boxes are manually placed at the start of conveyor A and then moved towards the pick & place workspace. Once within the reach of the pick & place, they are transferred to conveyor B, which moves them to the end of the line where they are manually removed.

### 3.4.2 "Automation devices" Classes

In order to program the system, it is important to identify the necessary classes. For this example, only two classes are needed since there are only two types of objects to control: the pick & place and the conveyors.

Figures 3.8 and 3.9 present the class diagrams for the "Conveyor" class, responsible for controlling the conveyors, and for the "PickPlace" class, responsible for controlling the pick & place, respectively.

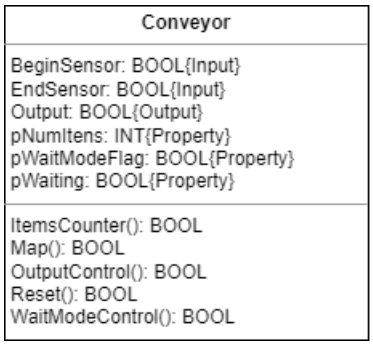



Figure 3.8: "Conveyor" class diagram

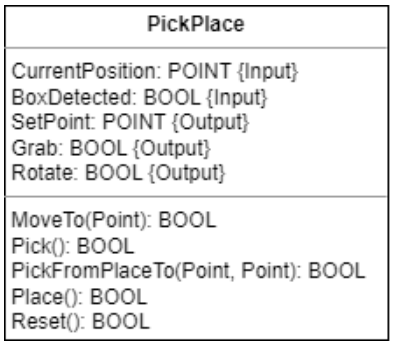



Figure 3.9: "PickPlace" class diagram

Figure 3.10 presents the project tree in CODESYS for the described system, including the PickPlace and the Conveyor classes.

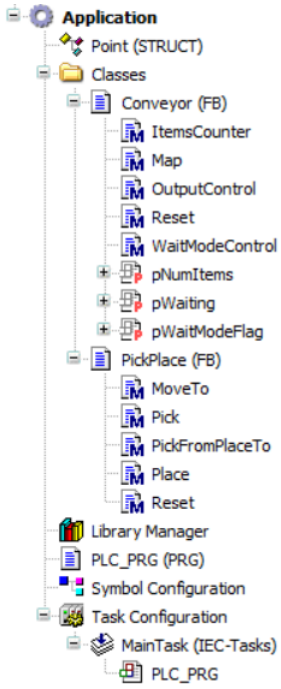



Figure 3.10: Pick & Place system project tree

#### 3.4.2.1 The PickPlace Class

The "PickPlace" class receives input from all sensors associated with the pick & place system and provides output to all associated actuators. Without including code in the body, the class controls the pick & place through methods, with each method being responsible for commanding one behavior of the equipment.

In order to simplify the definition of the inputs and the use of the developed methods, a structure named "Point" was created. A structure works basically as a variable type created by the user and, in this case, it comprises three real numbers (X, Y, and Z) that define a point in space. Therefore, instead of having, for example, to input three values to define the destination of the pick & place the user just has to input a "Point" structure as seen in the Figure 3.9.

The methods used in this example are described as follows:

- The method "MoveTo" takes a "Point" structure as the input and is responsible for moving the pick & place head to that point;
- The method "Pick" has no inputs and is responsible for lowering the pick & place head so that it can grab the box and then raising the head again;
- The method "Place" has no inputs and is responsible for lowering the pick & place head so that it can release the box and then raising the head again;
- The method "PickFromPlaceTo" takes two "Point" structures (the initial box location and the destination) as the inputs and, using the three previously defined methods, thanks to encapsulation, is responsible for picking the box from its initial position and placing it at the specified destination;
- The method "Reset" has no inputs and is responsible for getting the pick & place to its default state.

Figure 3.11 presents the mentioned methods with their respective code and required inputs.

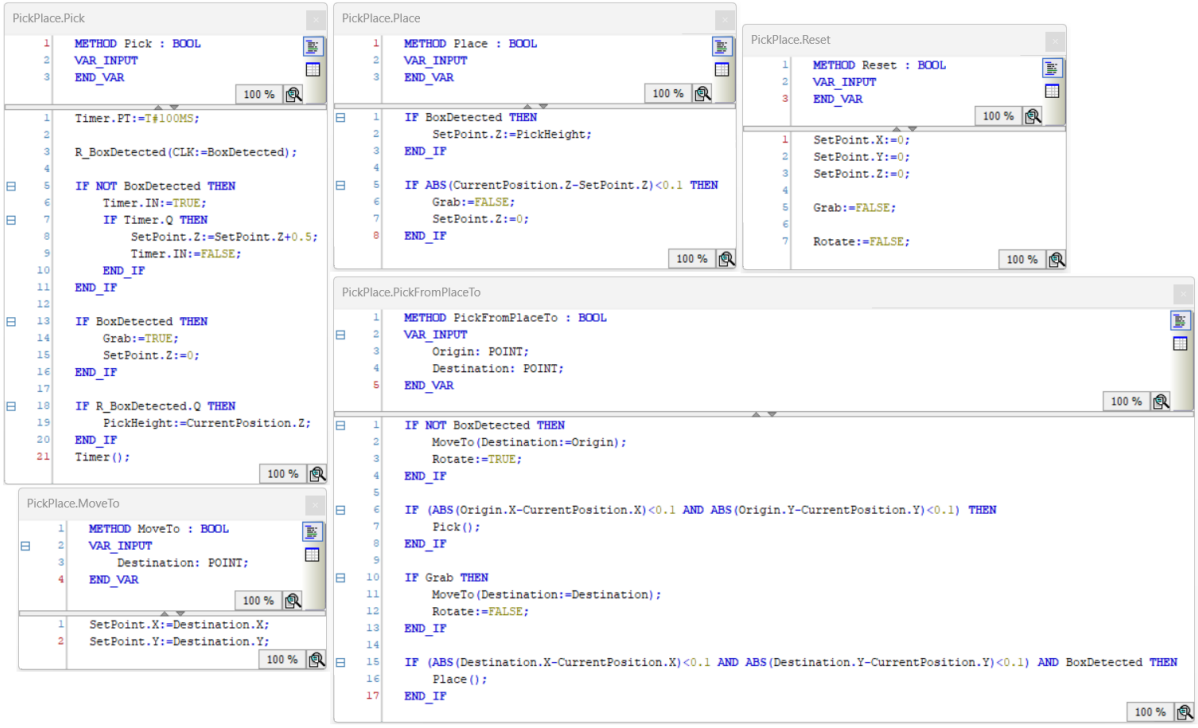



Figure 3.11: Methods of the "PickPlace" class

In addition to these methods, the class also features two properties: "pHasBox", which is a boolean that identifies when the pick & place is carrying a box and "pCurrentPos", which is a "Point" structure that contains the current position of the pick & place head.

Additionally, many others properties and methods could be defined by simply adding them and defining the associated algorithms, without needing to modify the existing structure or having to add new inputs.

### 3.4.2.2 The Conveyor Class

The "Conveyor" class receives input from sensors placed on the conveyor (one at the beginning of the conveyor - "BeginSensor" - and one at the end of the conveyor - "EndSensor") to identify when items enter or leave the conveyor.

The methods used in this example are described as follows:

- The 'ItemsCounter' method is responsible for counting the entrances and exits of items, updating the 'pNumItems' property.
- The "Map" method is responsible for defining rising edges, falling edges, and other necessary variables for the code;
- The "WaitModeControl" method is responsible for controlling the wait mode. If the "pWaitModeFlag" is true, the conveyor enters wait mode. In this mode, the conveyor stops when an item reaches the end sensor, waiting for the item to be removed or for the wait mode to be turned off before resuming movement;
- The "OutputControl" method controls the output of the conveyor considering the "pNumItems" property and the "pWaiting" property, which identifies when an item is detected by the end sensor in wait mode;
- The "Reset" method puts the conveyor in its default state.

Figure 3.12 presents the mentioned methods with their respective code and required inputs.

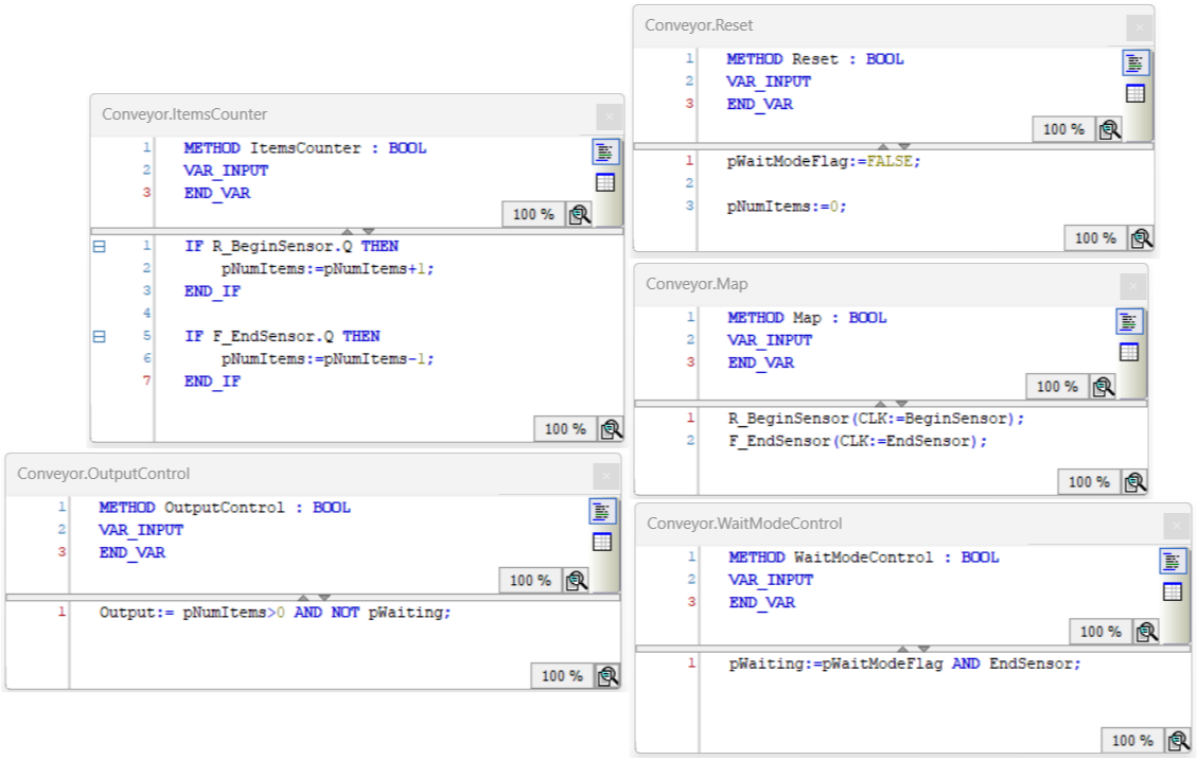



Figure 3.12: Methods of the "Conveyor" class

It is worth noting that, unlike the "PickPlace" class, the "Conveyor" class needs most of its methods to be constantly executed, and as such, these are called directly within the body of the class itself.

Additionally, as with the pick & place class, many other properties and methods can be defined by simply adding them and specifying the associated algorithms, without needing to modify the existing structure or add new inputs.

### 3.4.3 Supporting Interaction among "Automation" Objects

Having defined the classes, it is necessary to instantiate them in the PLC_PRG in order to create the objects that constitute the system and define the intended interactions. To ensure proper interaction between the system's objects, it is necessary, as mentioned in section 3.3, to have "management objects". In this example, two handshakes would be necessary: one between conveyor A and the pick & place, and another between the pick & place and conveyor B.

To create these handshakes, the class "Handshake" defined in section 3.3.1 can be used. In order to access its methods and properties, it is necessary to define a new input/output variable ("VAR_IN_OUT" in CODESYS) of the type "Handshake" in both the "PickPlace" and "Conveyor" classes.

Using both handshakes, it is possible to ensure that the pick & place only initiates the transportation of a box when there is a box at the end of conveyor A and conveyor B is available to receive it. Figure 3.13 represents the sequence diagram of the system operation considering that a box is introduced into conveyor A while the system is empty.

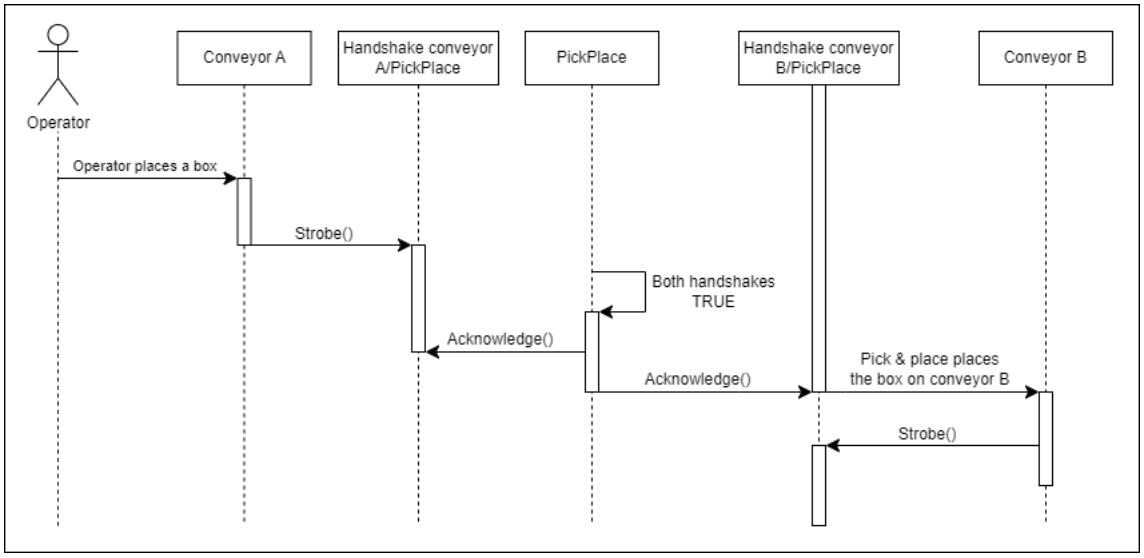



Figure 3.13: Sequence diagram of the operation of the system

The strobe of both handshakes is controlled by their respective conveyors: the "Strobe()" of handshake A is called when a box is detected at the end of conveyor A, and the "Strobe()" of handshake B is called when a box is no longer detected at the beginning of conveyor B.

The acknowledgments of both handshakes are called by the pick & place: the "Acknowledge()" of handshake A is called when the pick & place removes the box from conveyor A, and the "Acknowledge()" of handshake B is called when the pick & place places the box on conveyor B.

If an additional input conveyor were integrated into the system, it would be necessary, in addition to another handshake, to have a queue so that the boxes could be transported in the order they arrive. To do so, it would be sufficient to add to the system a queue like the one defined in section 3.3.2, so that, according to the arrival order of the boxes, it would be possible to define which movement the pick & place should perform.

## 3.5 Chapter summary

Considering that object-oriented programming was originally used in information systems, this chapter focus was to evaluate to what extent the application of OOP in automation systems could follow in the footsteps of information systems and thus inherit the proven good results. For that, a practical analysis of OOP was conducted, starting by distinguishing information technology from operational technology and introducing UML as the new modeling tool used with OOP.

Comparing information and automation systems, it was possible to conclude that although the two types of systems have very different goals, they end up using objects that, despite being different, essentially serve the same purpose. Thus, despite solving different problems, they end up approaching them in very similar ways. This emphasizes the potential for the industrial programming domain to draw inspiration from the successes of information systems programming, suggesting the feasibility of a comparable evolution to achieve comparable advancements.

However, the use of OOP in these two types of systems also presents several differences: information systems rely on synchronous programming and essentially use functions that are reentrant, while automation systems rely on asynchronous programming and essentially use functions that are non-reentrant. To address these differences, two elements for managing object interaction were introduced: queues, which allow automation systems to maintain con-

currency despite being essentially non-reentrant, and handshakes, which are used to synchronize objects in automation systems when needed due to their asynchronous programming.

Finally, the implementation of some of the presented concepts in CODESYS was introduced using a conceptual example of an industrial system.

In the next chapter, a case study will be analyzed, detailing a system initially programmed using procedural programming and then reprogrammed using OOP. The objective is to compare both programming paradigms in order to validate the ideas presented throughout this document.

# 4 A Case Study on going from Procedural to Object-Oriented PLC Programming

In this document, OOP has been studied, from the fundamental principles that support it, such as encapsulation, abstraction, inheritance, and polymorphism, to the associated concepts, including classes, objects, and properties. Additionally, new modeling tools that have emerged alongside OOP, such as UML, have been introduced.

Furthermore, the advantages of this type of programming compared to procedural programming have been highlighted. OOP allows for the control of complex systems in a way that is much easier to interpret, modify, and expand by dividing them into objects that operate almost independently, interacting with each other to achieve the system's final goal.

Therefore, with a solid theoretical foundation established, this chapter presents a case study with the goal of validating the conclusions drawn throughout the document. The case study involves reprogramming a system, originally developed using procedural programming, into an object-oriented approach, identifying the advantages of this adoption when compared to the original program.

## 4.1 Presenting the case study

To demonstrate the practical application of OOP in an industrial system, a virtual system originally programmed in SFC will be used. The software used to simulate the system was Factory IO [7], which is a popular 3D factory simulation software that allows the creation and control of virtual scenarios in a simulation environment.

The system chosen was the central theme of the dissertation "Diseño de un sistema de control distribuido usando Factory IO y Codesys V3 (comunication through OPC)" by José Marín Sánchez in 2018 [27].

To begin the system analysis, a brief description is provided based on the original document. It is worth noting that, for the sake of clarity and focus, some minor systems details stated in the original document are ignored here [27].

### 4.1.1 Plant description

The system consists of two palletizing stations and an output conveyor-based routing system. Each palletizing station receives standard pallets and boxes of different sizes from different conveyors. A pick & place unit picks up the boxes, one by one, and places them on the pallets so that each pallet gets a box. After palletization, the palletized boxes are routed to specific exits according to their size through a maze of roller conveyors and turntables, which guide each box along a particular path toward its destination exit.

Therefore, the main components of the system are:

- Roller conveyors, responsible for transporting pallets (with and without boxes);
- Belt conveyors, responsible for transporting boxes;
- Pick & place units, responsible for transferring the boxes to the pallets;
- Turntables, responsible for orienting the boxes to reach the correct destination.

Additionally, there are numerous sensors that allow the entire system to be controlled.

Figure 4.1 represents the original system and Figure 4.2 represents an equivalent system developed for the purpose of this dissertation.



Figure 4.1: Original system [27]



Figure 4.2: Developed system

Figures 4.3 and 4.4 name the elements that make up the system.

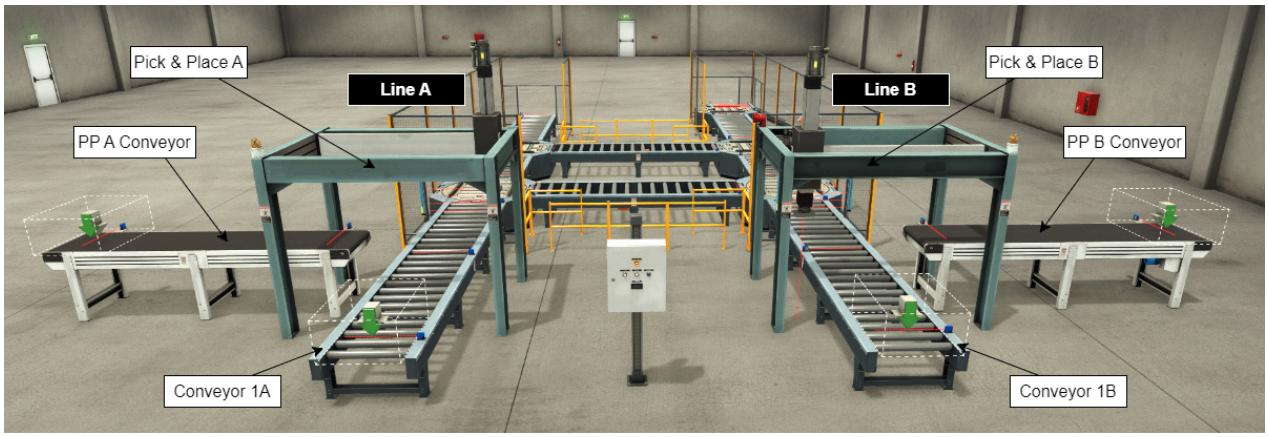



Figure 4.3: Systems elements labeled - palletizing zone

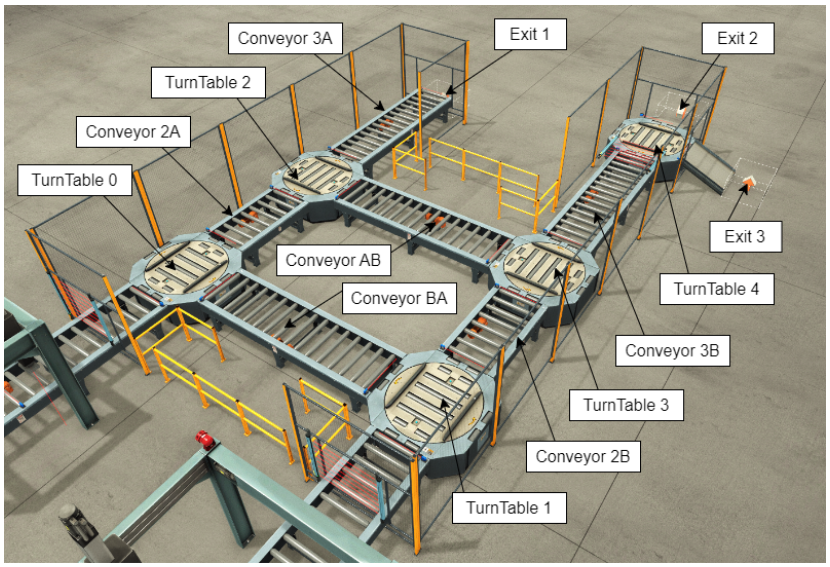



Figure 4.4: Systems elements labeled - routing/enclosed area

As can be seen, two main lines are distinguished in the system: line A on the left and line B on the right. These lines extend from the inputs to inside the enclosed area.

### 4.1.2 Plant detailed operation

As mentioned, the system's goal is to palletize boxes and then route them to specific exits based on their size. The boxes are manually placed on the "PP A Conveyor" and "PP B Conveyor" belts, which are responsible for transporting them within the reach of the respective pick & place. Figure 4.5 shows the three types of boxes that can exist in the system.

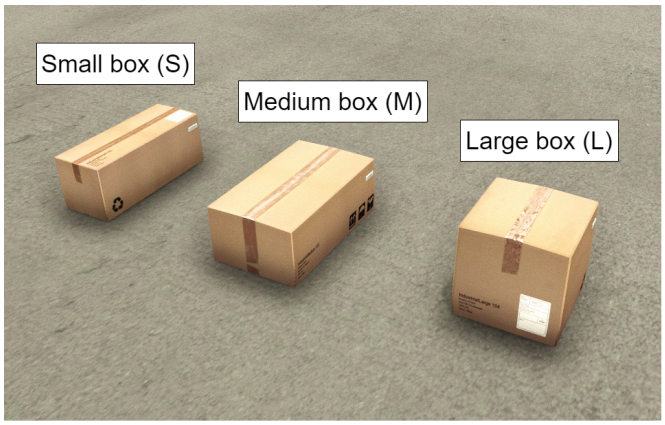



Figure 4.5: Types of boxes

In conveyors 1A and 1B, pallets are manually introduced and then moved along until they reach the base of the belt conveyors carrying the boxes. When there is a box at the end of the belt conveyors and a pallet in front of it, the pick & place transports the box to the pallet. After the pick & place releases the box, the pallet-box assembly is transported into the enclosed area of the system where the actual separation will take place.

Upon entering the enclosed area, the pallet with the box crosses a photoelectric barrier that allows for the identification of the box height, thus distinguishing between L-sized boxes and S/M-sized boxes.

Inside the enclosed area, the turntables are responsible for adjusting their orientation according to the size of the box, ensuring it reaches the intended exit. However, considering that the photoelectric barrier at the entrance of the enclosed area only allows for distinguishing boxes based on their height, it is not possible to differentiate between S and M boxes. For this purpose, another horizontal photoelectric barrier is placed before turntable 4, allowing for the distinction of boxes based on their width.

Thus, the L boxes are sent to exit 1, while the others are directed towards turntable 4, where the distinction between S and M boxes is made, sending M boxes to exit 2 and S boxes to exit 3.

To better explain the system's operation, Figure 4.6 shows the sequence of movements of the turntables and conveyors in order to organize one box of each type (S, M, and L).

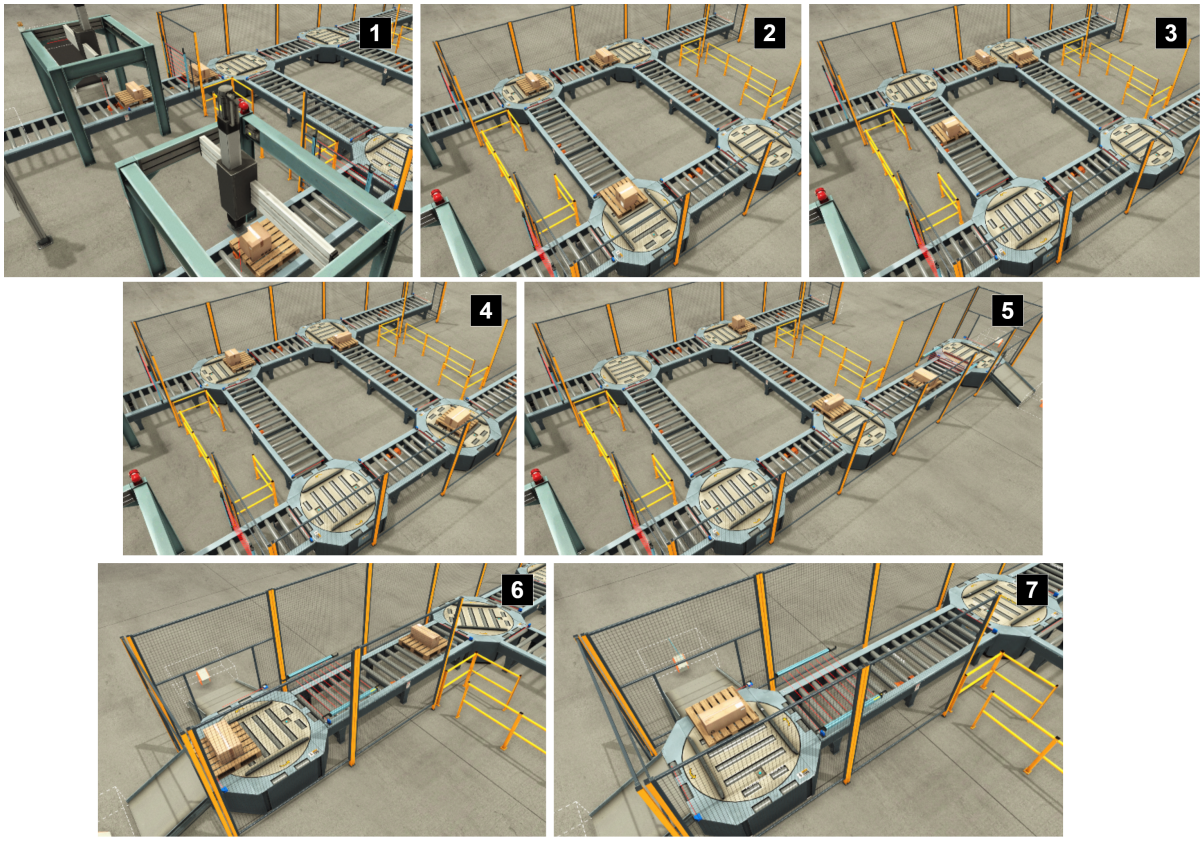



Figure 4.6: Representation of the system's operation

### 4.1.3 User interface

User interaction, is carried out through buttons, signaling lights and a numerical display located on an electrical panel shown in the Figure 4.7.

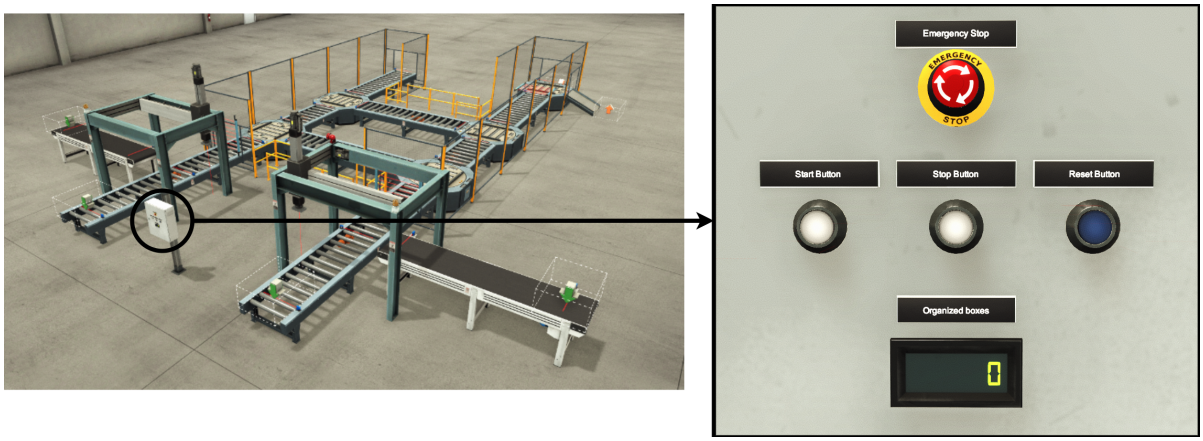



Figure 4.7: Electrical HMI panel

Pressing the start button turns the system on, making it ready to receive boxes and pallets. While the system is running, the numerical display shown in Figure 4.7 shows the total number of routed boxes, that is to say, the number of boxes that have already left the system.

At any time during the system's operation, pressing the stop button puts the system into a "plant closing" state where the boxes already in the enclosed area are routed, but no new

boxes are placed onto pallets, stopping the conveyors 1A and 1B as well as the pick & places. Once the enclosed area is clear of boxes, the system becomes idle, waiting for the start button to be pressed again.

To ensure safety, there is an emergency stop button. Pressing this button stops system operation and triggers a loud alarm. To reset the emergency state, the emergency stop button has to be released, allowing the reset button to be effective. After this, the system enters the reset mode, moving all boxes within the system to the nearest exit, regardless of their size. This clears the system, allowing for a clean start.

In the original version of the system, there is also a two-position turn switch that allows automatic or manual operating modes to be selected. However, within the scope of this document, only the automatic mode has been considered, as the manual mode would only allow manual activation by the users of each system element.

For a more in-depth description of the system, reference [27].

## 4.2 Original procedural programming solution

Having analyzed the function of the system, its elements, and its operation, the original solution, developed in SFC using procedural programming, will now be briefly presented.

SFC is a programming language that closely resembles GRAFCET, with programs structured around states and the transitions between them, thus functioning sequentially.

The original program structure is represented in Figure 4.8.

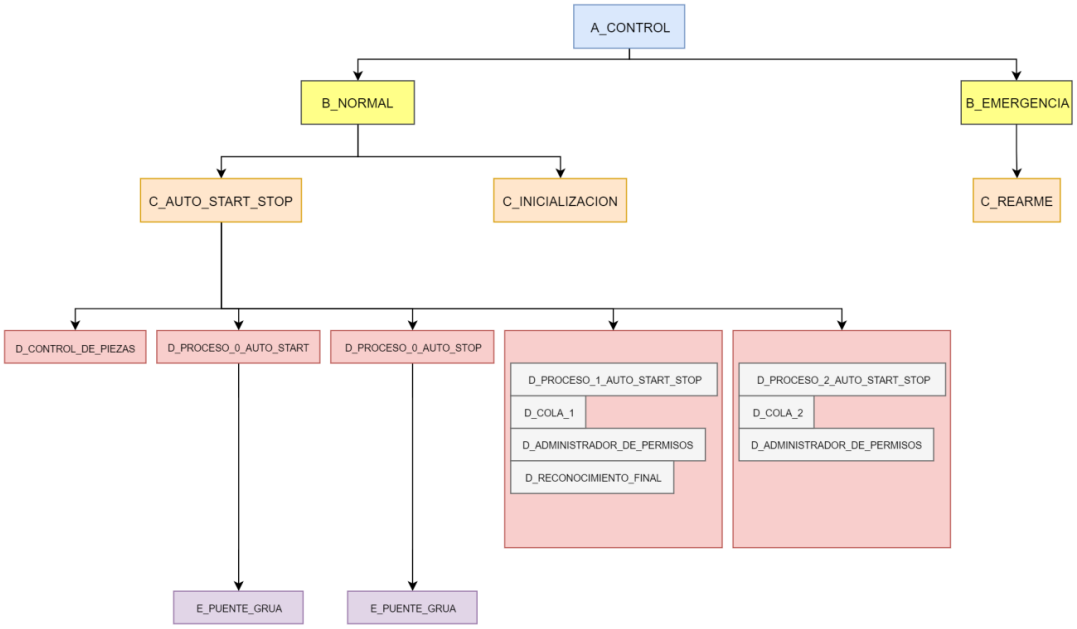



Figure 4.8: Schematic representation of the original solution [27]

Figure 4.8 presents the various programs that compose the system and how they relate to each other. Considering that most of the programs were developed in SFC, one can think of the programming of this system as hierarchical grafcets, where grafcets at a higher level call grafcets at lower levels to subdivide the program into parts.

The programs at the top function as high-level grafcets, meaning grafcets that are at the top of the hierarchy. As such, they are executed first and therefore are responsible for defining in which state the system is at each moment, invoking other programs responsible for defining the behavior of that state.

These programs called by the high-level grafcets, subsequently, occupy a lower position in the hierarchy and are responsible for controlling the system invoking more deterministic programs, developed in ladder or ST, which directly define the system variables in order to achieve the desired behavior.

In this way, to control the system, all possible states were manually defined, and through hierarchical relationships between grafcets, the correct programs are called at each instant: the GRAFCET at the top of the hierarchy is responsible for deciding in which state the system is, invoking another GRAFCET one level below in the hierarchy capable of defining that state by calling Ladder or ST programs that define the system variables in order to achieve the desired behavior.

To clarify this, the next section will present as an example the hierarchy of grafcets responsible for controlling the normal automatic mode of the system.

### 4.2.1 Procedural programming of complex hierarchical relations

To better demonstrate how the original solution is structured, the hierarchy of programs composed by "A_CONTROL", "B_NORMAL" and "C_AUTO_START_STOP" responsible for defining the normal automatic mode of the system will be presented very succinctly.

#### A_CONTROL Program

The program "A_CONTROL", modeled in Figure 4.9, is developed in SFC language and serves as the central control program of the system. It acts as the highest-level GRAFCET, initiating all hierarchies within the system. It is responsible for transitioning between normal operation mode and emergency mode by invoking the "B_NORMAL" program or the "B_-EMERGENCIA" program, respectively.

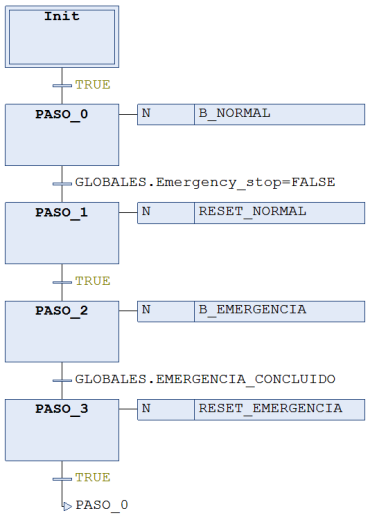



Figure 4.9: Program "A_CONTROL" [27]

#### B_NORMAL Program

The program "B_NORMAL", modeled in Figure 4.10, is developed in SFC language and is the second on this hierarchy. It is responsible for ensuring good interaction between the automatic and manual modes of the system. To achieve this, it runs an initialization procedure through the "C_INITIALIZATION" program before entering the automatic mode, ensuring that the automatic mode functions correctly even after using the manual mode. After running

the initialization procedure, the automatic mode is initiated, being driven by the "C_AUTO_-START_STOP" program.

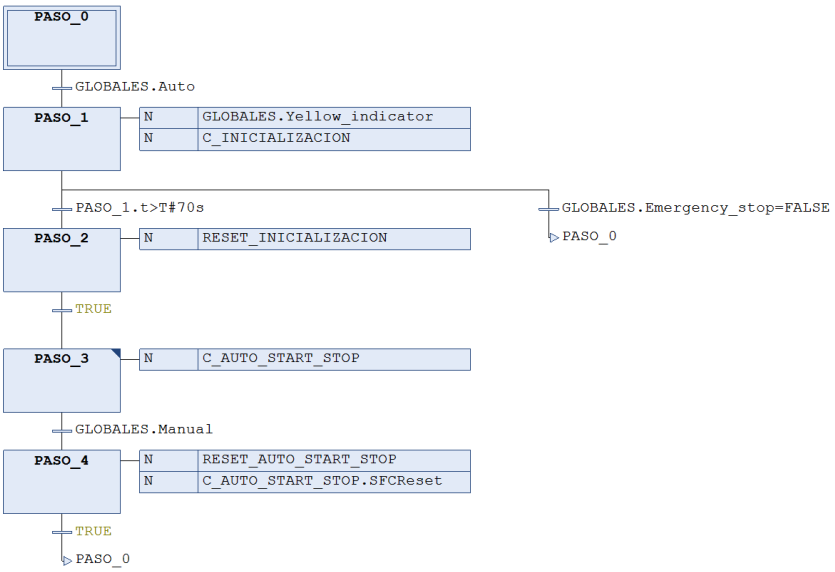



Figure 4.10: Program "B_NORMAL" [27]

### C_AUTO_START_STOP Program

The program "C_AUTO_START_STOP", modeled in Figure 4.11, is developed in SFC and is the last in this hierarchy. It is responsible for defining the intended behaviour for the normal automatic mode of the system just like a normal GRAFCET. For that, it uses other programs, developed in Ladder or ST, that directly control the elements of the system.

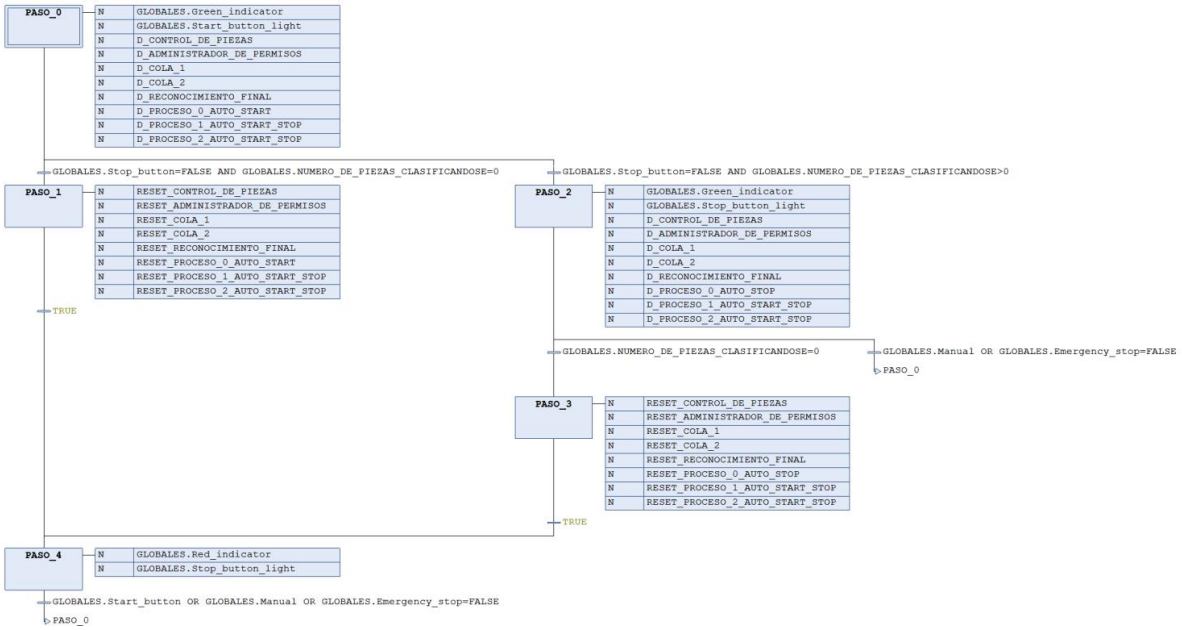



Figure 4.11: Program "C_AUTO_START_STOP" [27]

One of the programs used by the "C_AUTO_START_STOP" program is the "D_PRO-CESSO_1_AUTO_START_STOP" program, which is modeled in Figure 4.12, illustrating its complexity without providing exhaustive details.

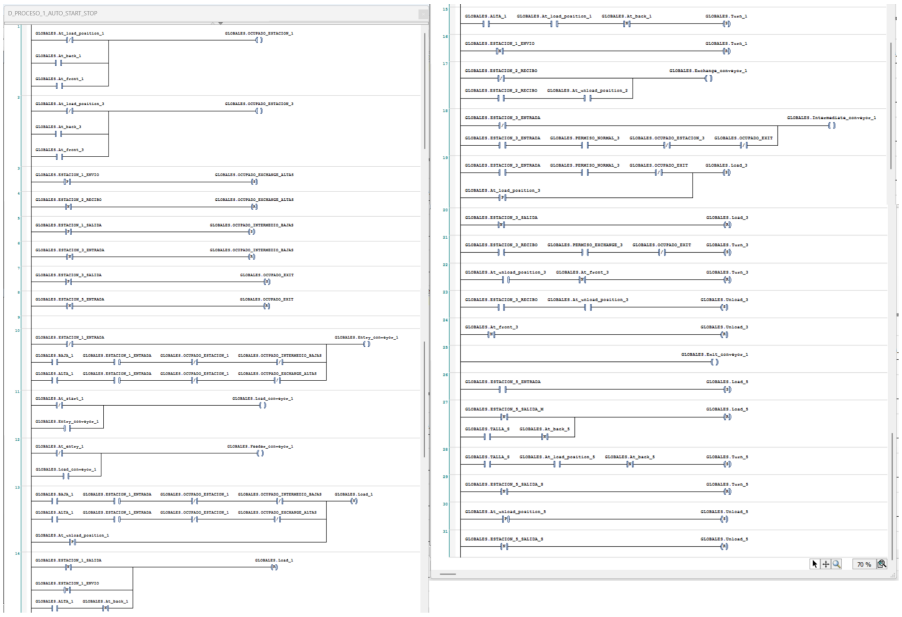


Figure 4.12: Program "D_PROCESSO_1_AUTO_START_STOP" [27]

### 4.2.2 Critical appraisal

Through this relatively brief analysis of the original system programming, which is not the main focus of this chapter, it can be concluded that, being essentially programmed in SFC and composed of various programs, it can be interpreted as hierarchical grafcets that work sequentially, therefore following a procedural programming approach.

In line with procedural methodologies, the original program defines every possible state for the system in different programs that are then called by higher-level programs that determine the current state of the system. Therefore, the programs used are not instantiated, executing as subroutines.

Even though the program is divided into many smaller programs to make it easier to understand, the programs that directly define the variables of the system, such as the one represented in Figure 4.12, are notably lengthy and complex, making them difficult to interpret and, therefore, to alter.

Furthermore, the fact that programs invoke other programs goes against IEC 61131-3 recommendations and adds complexity to the system, especially in larger systems like the one presented, where discerning which program invokes which, which programs are running, and for what purpose becomes challenging.

Additionally, the sequential nature of the code makes it exceedingly difficult to make any changes, as modifying one state will likely have implications on the following states, leading to a kind of "snowball effect" where even a small change could demand a complete rewrite of the code.

This leads to the conclusion that this type of programming ultimately proves very challenging to interpret and consequently difficult to modify, turning the program into a "black box", where only the developer who created it can understand and, with considerable effort and patience, modify it.

In order to show that using OOP this system can be programmed in a much easier and straightforward way, the next section presents the OOP solution developed as part of this document.

## 4.3 Designing and programming a Object-Oriented solution

As part of this document, a new PLC programming solution was developed from scratch to address the same problem. This solution aims to demonstrate how using OOP simplifies the system's programming, making it easily modifiable, expandable, and much easier to interpret.

### 4.3.1 Taking OOP as a new mindset

Before diving into the actual programming, it is necessary to first consider how to approach and model the problem using OOP.

In terms of the objects needed to control the system, it is clear that conveyors, turntables, pick-and-place units, and queues are required. Considering these objects and the goal of the system, there are essentially two main operations to address: moving the boxes to the pallets and routing the palletized boxes.

The transportation of boxes to the pallets results from the interactions between the pick & place units and the conveyors 1A and 1B. Figure 4.13 shows a sequence diagram that illustrates the intended operation, using the example of moving a box from "PP B Conveyor" to "Conveyor 1B".

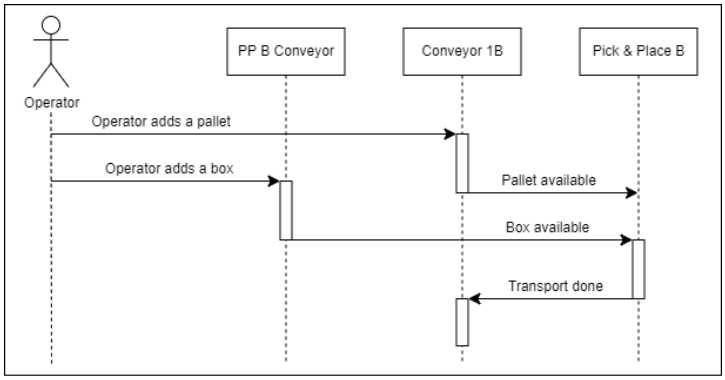



Figure 4.13: Sequence diagram of the transport by the pick & places

The routing of the palletized boxes results from the interactions among the conveyors, the turntables, and the queues. Figure 4.14 shows a sequence diagram that illustrates the intended operation, using the example of a large box entering the enclosed area on line B and heading towards line A.

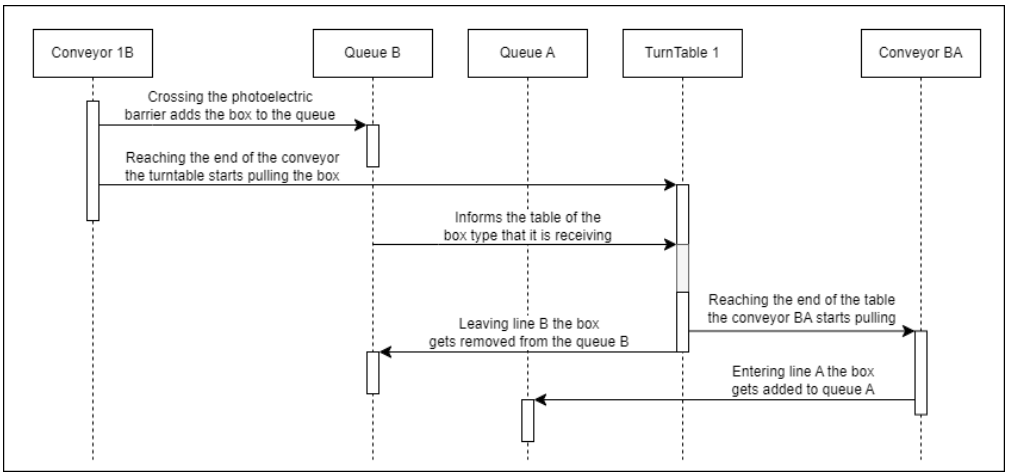



Figure 4.14: Sequence diagram of the transport by the turntables

Having the system's core operations outlined, programming can now proceed by developing the necessary classes to achieve the desired functionality.

### 4.3.2 Classes representing "Automation Devices" and "Management Resources"

To define the objects that constitute the system, it is necessary to delineate the classes. For this system, the following four classes were considered:

- The "VerifierQueue" Class - responsible for managing the system queues;
- The "PickPlace" Class - responsible for controlling the pick & place devices;
- The "Conveyor" Class - responsible for controlling the conveyors;
- The "TurnTable" Class - responsible for controlling the turntables.

#### 4.3.2.1 The VerifierQueue Class

In the system, there are two obvious queues: one responsible for keeping track of the boxes on side A, QueueA, and another one for keeping track of the boxes on side B, QueueB. Through these queues, it is possible to identify how many boxes are on each side and the type of each box so that when it reaches a turntable, it can position itself correctly, considering the box type.

Each of these queues has two associated devices: the photoelectric barrier, which functions as an initial sensor by adding the respective boxes to the queue when they enter the enclosed area, simultaneously allowing the identification of the box type; and the end sensor, which detects the exit of the boxes from the system so that they can be removed from their respective queue.

The class diagram and the code tree for this class are presented in Figures 4.15 and 4.16, respectively.

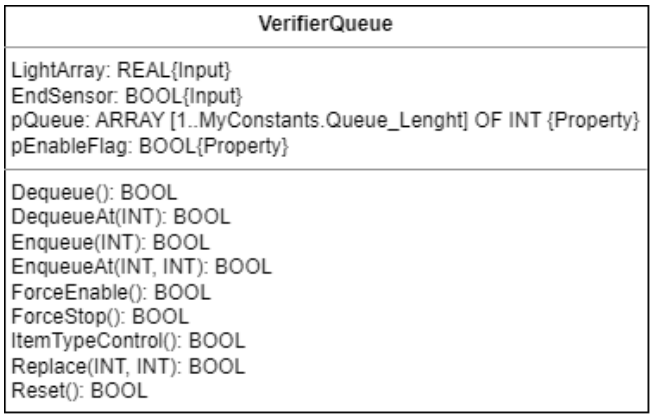



Figure 4.15: "Queue" class diagram

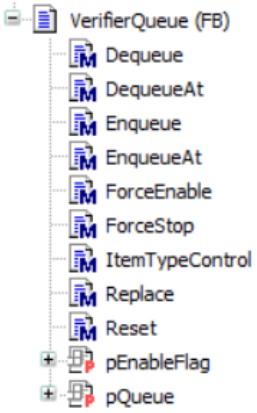



Figure 4.16: "Queue" class in CODESYS

This class is very similar to the one described in section 3.3.2, with only a few modifications:

- Two new inputs were added: "LightArray", which receives the value from the photoelectric barrier, and "EndSensor", which allows for identifying the exit of the boxes from the system;
- To interpret the value from the photoelectric barrier, the "ItemTypeControl()" method was added, which defines the type of box through the "LightArray" variable;
- In order to be able to disable the queue, the "pEnableFlag" property was added, which when FALSE inhibits all methods. To control this property, the "ForceEnable()" and "ForceStop()" methods were added.

#### 4.3.2.2 The PickPlace Class

The "PickPlace" class was developed to control the two pick & place devices of the system. This class is exactly the same as described in section 3.4.2.1, with the addition of the "pEnableFlag" property and the "ForceEnable()" and "ForceStop()" methods, just like for the "VerifierQueue" class, allowing to enable/disable the pick & place.

Figures 4.17 and 4.18 present the class diagram and the code tree for this class.

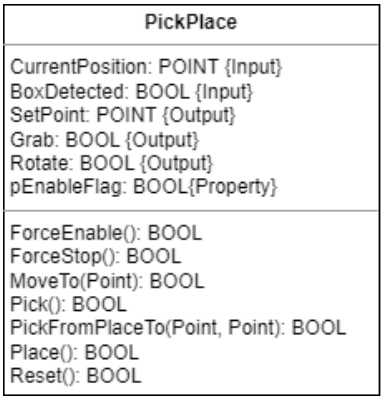



Figure 4.17: "PickPlace" class diagram

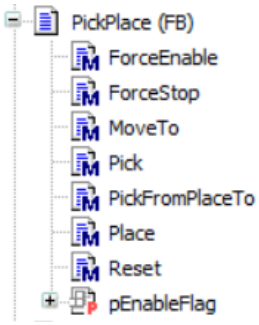



Figure 4.18: "PickPlace" class in CODESYS

#### 4.3.2.3 The Conveyor Class

The "Conveyor" class was developed to control the conveyors of the system. This class is exactly the same as described in section 3.4.2.2, with the addition of the "pEnableFlag" property and the"ForceEnable()" and "ForceStop()" methods, just like for the "VerifierQueue" and "PickPlace" classes, allowing to enable/disable the conveyor.

Figures 4.19 and 4.20 present the class diagram and the code tree for this class.

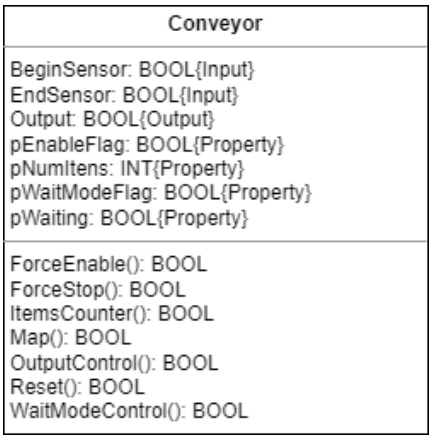



Figure 4.19: "Conveyor" class diagram

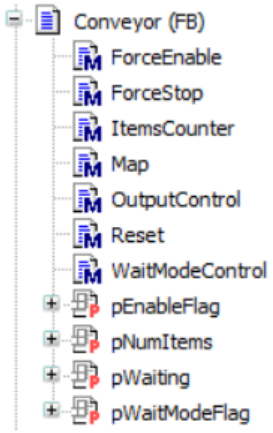



Figure 4.20: "Conveyor" class in CODESYS

#### 4.3.2.4 The TurnTable Class

The "TurnTable" class was developed to control the turntables of the system. To maintain simplicity, it was considered that each turntable has two entrances and two exits, as represented in Figure 4.21.

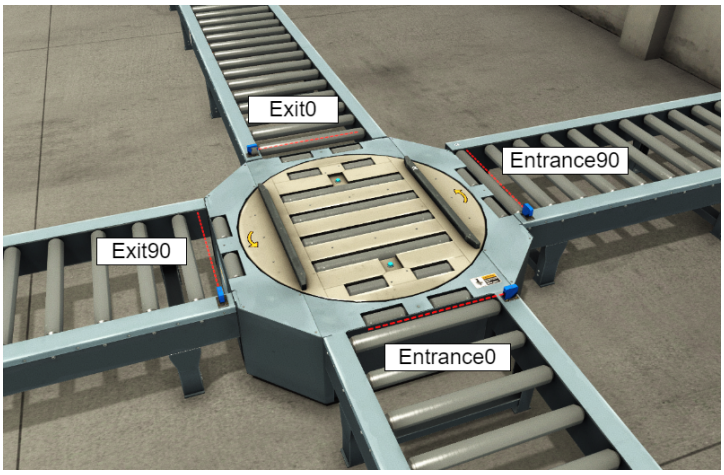



Figure 4.21: Turntable scheme

The entrance/exit 0 can only be accessed with the table in the default position (0 degrees, as represented in the Figure 4.21), while the entrance/exit 90 can only be accessed with the table rotated 90 degrees.

Figures 4.22 and 4.23 present the class diagram and the code tree for this class.

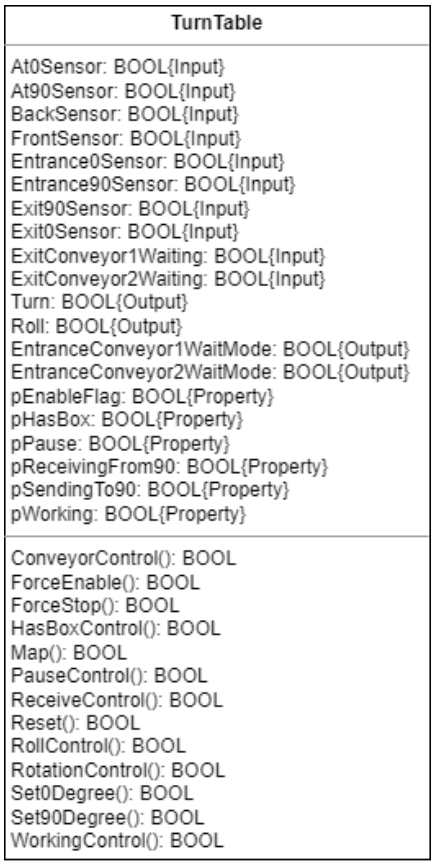



Figure 4.22: "TurnTable" class diagram

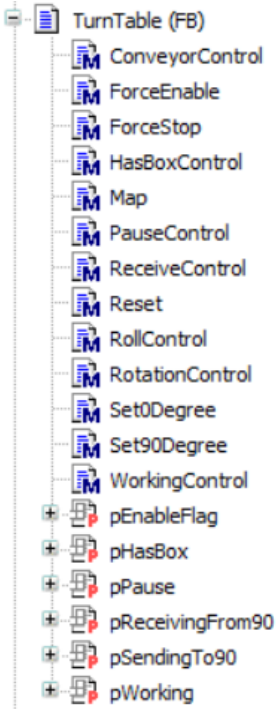



Figure 4.23: "TurnTable" class in CODESYS

In addition to receiving inputs from its own sensors and providing output to all its actuators, the "TurnTable" class also includes additional inputs and outputs to facilitate the interaction with the conveyors that directly engage with it.

As extra inputs, the turntables receive input from the sensors placed at the interface between the table and the conveyors and also receive input from the state of the output conveyors, identifying whether they are in a "Waiting" state or not. These additional inputs allow the table to correctly identify when to start/stop its operation.

As extra outputs, the "TurnTable" class has dedicated outputs for defining the state of the input conveyors. Therefore, when occupied, the table can set the input conveyors to "Wait mode".

The properties defined for this class are described as follows:

- pHasBox - Boolean property that identifies if the table contains a centered box (between the back sensor and the front sensor);
- pPause - Boolean property that pauses the table's roll if the destination conveyor is in a waiting state;
- pReceivingFrom90 - Boolean property that identifies if the table is set to receive from entrance 90. If false, the table is set to receive from entrance 0;
- pSendingTo90 - Boolean property that identifies if the table is set to send to exit 90. If false, the table is set to send to exit 0;
- pWorking - Boolean property that identifies if the table is working, turning to true whenever a box is detected near the table and returning to false whenever a box leaves the table.

In addition to these specific properties, the "TurnTable" class also has the "pEnableFlag" property, which has been described for the previous classes.

The methods defined for this class are described as follows:

- HasBoxControl(), PauseControl(), and WorkingControl() are responsible for controlling the properties pHasBox, pPause, and pWorking respectively;
- ConveyorControl() is responsible for controlling the wait mode of the input conveyors;
- RollControl() is responsible for controlling the roll of the table, causing the boxes on top of it to advance (or not);
- RotationControl() is responsible for positioning the table correctly to send/receive boxes;
- ReceiveControl() is responsible for controlling the property pReceivingFrom90, identifying the position at which the table should be placed to receive the next box;
- Set0Degree(), Set90Degree() are responsible for positioning the table at 0 and 90 degrees, respectively, relative to its default position (represented in Figure 4.21).

In addition to these methods, the "TurnTable" class also includes the methods "Map()", "Reset()", "ForceEnable()", and "ForceStop()", which are equivalent to the analogous methods used in the other classes of the system.

### 4.3.3 Interaction among objects

Having defined the classes that enable the creation of the objects that make up the system, it is now necessary to define the interactions between these objects. For this purpose, the defined classes were instantiated on the PLC_PRG to create the objects that will control the various elements of the system. Once the objects are defined, methods and properties can be created to manage their interactions, thereby controlling the system as a whole.

Figure 4.24 presents the methods and properties developed to manage the interactions between the objects of the system.

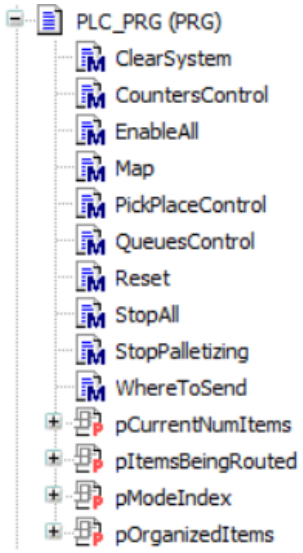



Figure 4.24: PLC_PRG in CODESYS

The developed methods are described as follows:

- ClearSystem() - Method responsible for configuring the system elements to eliminate all boxes in the system as quickly as possible - boxes are removed through the nearest exit regardless of their size;
- CountersControl() - Method responsible for controlling the counters used in the program;
- EnableAll() - Method responsible for activating the system by calling all "ForceEnable()" methods of all system elements;
- Map() - Method responsible for defining all the necessary rising and falling edges for the program;
- PickPlaceControl() - Method responsible for synchronizing the operation of the pick & places with the conveyors that transport pallets and boxes;
- QueuesControl() - Method responsible for controlling the two queues of the system;
- Reset() - Method responsible for resetting the system to the default state;
- StopAll() - Method responsible for deactivating the system by calling all "ForceStop()" methods of all system elements;
- StopPalletizing() - Method responsible for stopping the pick & places and the conveyors 1A and 1B, ensuring that no more boxes are introduced into the enclosed area;
- WhereToSend() - Method responsible for, using the queues, determining where each turntable should send each box considering its type.

Besides the methods, some properties were also defined to facilitate the control of the system. The properties created are described as follows:

- pCurrentNumItems - Identifies the total number of items currently in circulation within the system;
- pItemsBeingRouted - Identifies the number of boxes currently being routed inside the enclosed area;
- pOrganizedItems - Identifies the number of organized items, meaning the number of boxes that have already exited the system.

To define different behaviors for the system, five operating modes have been established:

- Stopped mode - System is idle;
- Transporting mode - System operates normally;
- Stopping mode - System stops putting new boxes into pallets and finishes routing the boxes in the enclosed area before transitioning to the stopped mode;
- EmergencyStop mode - System is in an emergency situation, therefore not operating;
- Resetting mode - System undergoes a reset after exiting the emergency mode, moving all boxes to the nearest exit (regardless of their size) to clear the system for a clean start.

Figure 4.25 shows the defined modes in CODESYS.

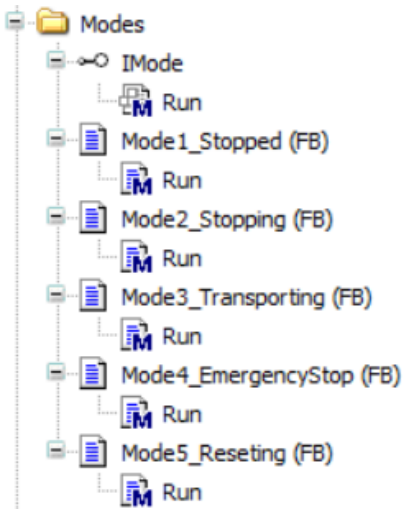



Figure 4.25: System modes in CODESYS

These modes are defined by implementing the "IMode" interface containing the "Run()" method, as shown in Figure 4.25. The PLC_PRG methods that must be executed within each mode to achieve the intended behavior are called from the "Run()" method. The transitions to other modes are also defined, by changing the "pModeIndex" property.

The code associated with each mode is shown in Figure 4.26.

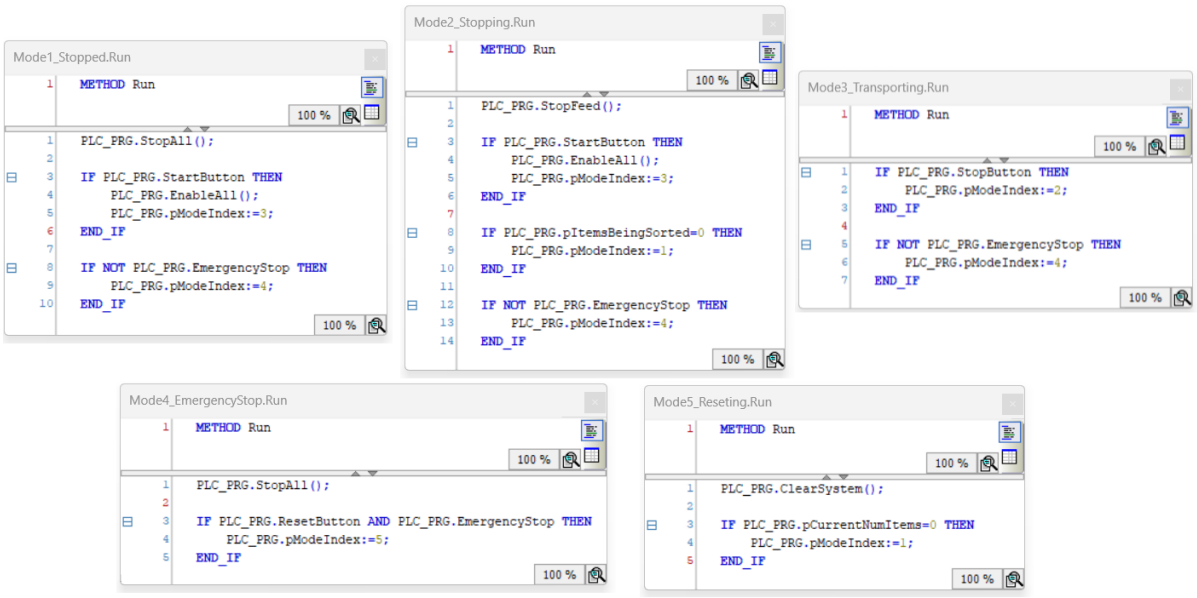



Figure 4.26: Code associated with each mode

All the other elements of the system that are not directly controlled by any object, such as the lights on the electrical HMI panel and the siren, are controlled by the PLC_PRG itself.

Figure 4.27 presents the UML component diagram of the system, illustrating the interactions between the "automation" objects that constitute it.

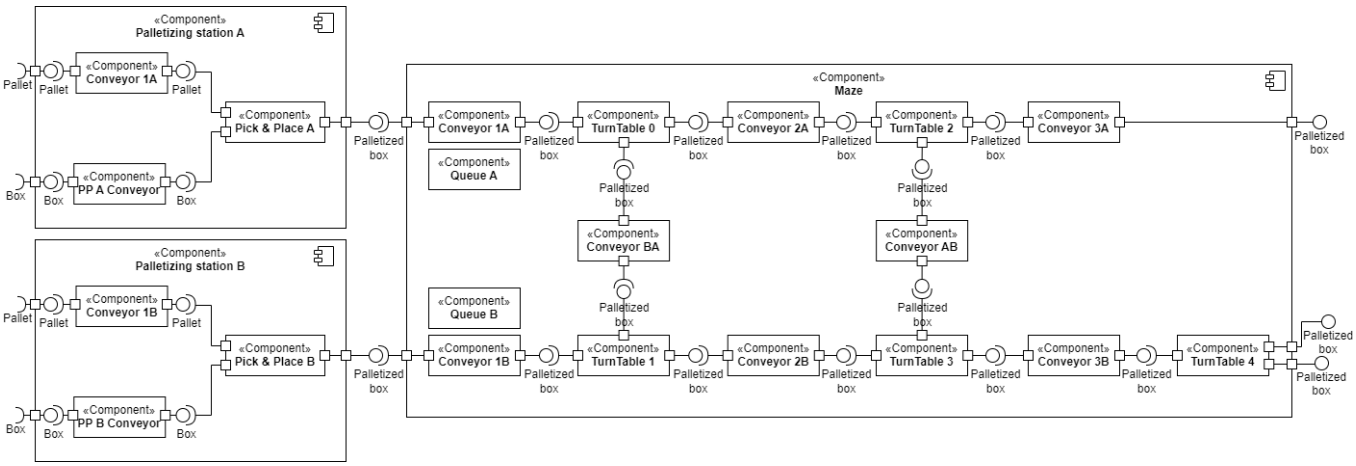



Figure 4.27: UML Component diagram of the system

### 4.3.4 Critical appraisal

Through OOP, the system is divided into small parts that function individually: the objects. Each system element corresponds to an object and operates on its own through the code defined in its class. These objects communicate with each other to collectively achieve the final goal of the system.

By dividing the system into small, self-contained units, OOP promotes modularity and flexibility by enabling code reuse and facilitating software changes, as a modification in a class affects all its objects without directly impacting the rest of the system. Additionally, the fact that it does not operate sequentially means that changes can be made to the system without significantly impacting its overall operation.

The system becomes easily expandable through modularity since new features can be added to the existing classes, and new objects can be easily created to expand the original system.

Furthermore, dividing the system into parts makes the code much easier to interpret since, instead of having programs with hundreds of lines, several small programs are used that, working together, achieve the intended goal.

Therefore, the developed program is much easier to interpret, far more flexible as changes can be implemented without reprogramming the entire system, and highly scalable since new features or objects can be added effortlessly.

## 4.4 Chapter summary

In this chapter, a case study was presented to validate the conclusions drawn throughout the document. The study involved reprogramming a system that was originally developed using procedural programming, now transitioning to object-oriented programming.

The original system, developed with SFC, employed complex and difficult-to-interpret code, which, operating sequentially, made it hard to alter and expand without impacting the overall system's functionality.

Using OOP, the system was reprogrammed, leading to a much more organized structure that promotes modularity by segmenting the system into objects and facilitates interpretation. Thanks to its modularity, independent operation of its elements, and non-reliance on sequential execution, the new program also becomes much easier to modify and expand.

Comparing the two programs, it becomes evident what was mentioned in section 2.3.1.3: the original program, using procedural programming, functions like a very complex mechanical clock, where all elements operate very rigorously and even a small alteration can disrupt the system completely, making it really hard to alter/expand; on the other hand, the OOP program functions like a living organism, where each object operates like a cell working together to ensure the intended functioning of the system, promoting modularity and making the system less rigid, facilitating its expansion and modification.

Through the case study, it was possible to demonstrate how impactful OOP can be in programming complex systems. The transition from a procedural programming based system to one built with OOP principles showcased not only improved organization and ease of interpretation but also highlighted the modularity and flexibility inherent in OOP. This shift not only simplifies system maintenance and expansion but also fosters a more dynamic approach to system development, ultimately leading to more efficient and adaptable software solutions.

In the next chapter, the concept of software libraries will be explained, addressing their purpose, advantages, and important aspects to consider when developing them. As an example, the sample class library developed as part of this document will be presented.

# 5 Development of a functional library for supporting Object-Oriented PLC Programming

As discussed throughout the document, object-oriented programming allows complex systems to be divided into smaller self-contained parts, the objects, which communicate with each other to achieve the system's final goal. This promotes modularity and flexibility, facilitating changes to the program and leading to more efficient and scalable solutions.

This independence between the different objects that make up the system allows each of them to be introduced into other systems and continue to function in the same way. In other words, since the classes developed for a particular program operate independently, each can easily be introduced into other programs so that they can be instantiated to control the elements of those programs, maintaining the original performance. It is based on these ideas that the concept of libraries was created.

In this section, the concept of libraries is discussed, identifying their advantages and important aspects to consider during their development. The class library developed in the context of this document is also presented.

## 5.1 Concept and purpose

In general programming, libraries are collections of prewritten code and other software elements that programmers can use to optimize tasks, address and support common problems. In object-oriented programming, the term "class library" is commonly used underlying the idea that, in this context, libraries are collections of classes.

By importing a class library, the classes defined within it become available for use in the program and can be directly instantiated to control the system's objects. In addition to containing classes, class libraries may also include other software elements such as interfaces, structures, user defined data types and variable lists. Furthermore, libraries can contain other libraries, meaning that a particular library may be developed using elements from another, which is immediately imported when the former is imported.

### 5.1.1 Advantages

Libraries provide numerous advantages in software development, some of which are addressed in this section.

By leveraging prewritten code, libraries constitute a base for programs, preventing programmers from having to start from scratch and allowing to integrate proven solutions into new projects, thereby streamlining the development process of applications and alleviating the burden of coding tasks. This efficiency not only accelerates project timelines but also reduces the overall workload on developers and ensures consistency and reliability.

Moreover, the collaborative nature of libraries fosters a culture of code-sharing among users. By enabling developers to share their code with others, libraries create a vast repository of solutions for multitude of common problems encountered in software development. This abundance of resources not only expedites problem-solving but also promotes innovation and knowledge exchange within the developer community.

Additionally, libraries serve as a protective mechanism for the developed code. By abstracting and encapsulating functionalities within library modules, developers guarantee author's rights and shield their codebase from unintended modifications, maintaining the integrity of their software solutions. This safeguarding mechanism ensures that the core functionalities remain intact, even as the application evolves over time.

### 5.1.2 Development aspects

For libraries to effectively fulfill their purpose, it is necessary for those who will use them to be able to understand how they work. Therefore, it is necessary to follow certain guidelines in their development to ensure that all users can understand and utilize them. These guidelines aim to prevent inconsistencies and ensure compatibility among different libraries [28].

According to CODESYS, some of the guidelines to follow are:

- Libraries should have meaningful names;
- Templates can be used to maintain consistency;
- A familiar and, whenever possible, consistent project structure should be employed;
- Project information such as title, version, company and category should be provided;
- User-friendly error handling should be implemented;
- A naming convention should be followed to ensure clean code;
- When modifying an existing library, interface compatibility with previous versions should be ensured.

Another essential element associated with library development is the documentation. The documentation associated with a library is a very important tool as it allows transmitting information to users about it. Through the documentation of a library, it is possible to inform users about, for example, the library's purpose, the problems it aims to address, how to use the library's elements and how to install the library. Therefore, facilitating the library's use for all users, ensuring that they have proper help and support when using it.

## 5.2 The developed class library

In the scope of this document, a class library centered on elements from the Factory IO collection was developed with the aim of providing the necessary tools for its users to create a wide variety of systems, including something similar to what was presented in section 4.3.

As such, the library includes classes very similar to those presented in section 4.3, along with additional classes that make it more versatile, allowing users to create more varied systems and even modify the system presented in chapter 4 by replacing some of its elements.

The library includes four classes that can be used to replicate the system from chapter 4, namely:

- Class "Queue" - responsible for controlling the queues in the system;
- Class "PickPlace" - responsible for controlling the two pick & place devices;
- Class "Conveyor" - responsible for controlling the conveyors;
- Class "TurnTable" - responsible for controlling the turntables.

As well as the following three new classes:

- Class "TwoAxisPickPlace" - responsible for controlling two-axis pick & places;
- Class "BidirectionalConveyor" - responsible for controlling bidirectional conveyors;
- Class "ChainTransfer" - responsible for controlling chain transfer tables.

It is important to note that the developed class library does not intend, in any way, to be an exhaustive and complete tool, facilitating the programming of every Factory IO scenario. It is merely, and obviously, a model resource, which, covering some important elements, aims to exemplify the functioning and development of a library.

### 5.2.1 The Queue Class

A class "Queue" has been included in the class library to allow users to replicate the queues described in section 4.3.2.1.

In order to give more freedom to users, it was decided not to adopt the "VerifierQueue" class presented in section 4.3.2.1. This is because that class was specifically developed for the system presented in chapter 4, receiving inputs from photoelectric barriers and having the possible item types defined within it through the "ItemTypeControl()" method. Therefore, it would be impossible for users to define other types of items or even change the way they are detected.

Instead, the decision was made to adopt the "Queue" class defined in section 3.3.2. This gives users the freedom to define item types outside of the class as they see fit, and to employ sensors other than photoelectric barriers.

Figures 5.1 and 5.2 present the class diagram and the code tree obtained for this class.

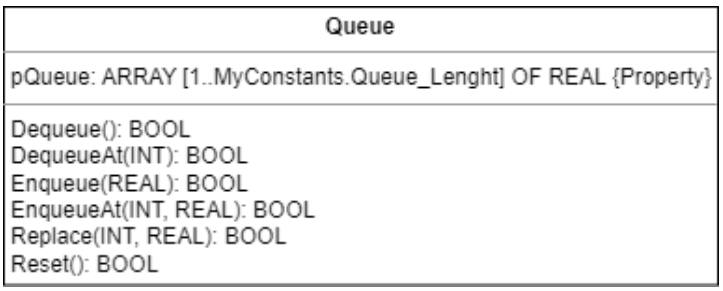



Figure 5.1: "Queue" class diagram

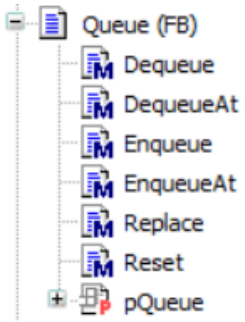



Figure 5.2: "Queue" class in CODESYS

### 5.2.2 The PickPlace Class

The "PickPlace" class was included in the class library to allow users to replicate the functionality of the pick & places described in section 4.3.2.2. To adapt it to the library, the following inputs were added to enable the control of pick & places without the need for additional code:

- "StartCondition" and "StopCondition" - Boolean inputs that allow users to enable and disable the pick & place without having to use the "ForceEnable()" and "ForceStop()" methods;
- "PickPoint" and "PlacePoint" - "Point" type inputs that allow users to define the pick location and the place location of the pick and place operation that can be initiated through the "RunCondition" input;
- "RunCondition" - Boolean input that allows users to initiate a pick and place operation from the "PickPoint" input to the "PlacePoint" input without having to use the "PickFromPlaceTo()" method.

Figures 5.3 and 5.4 present the class diagram and the code tree obtained for this class.

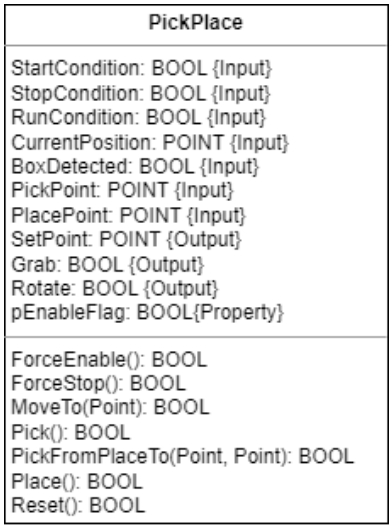



Figure 5.3: "PickPlace" class diagram

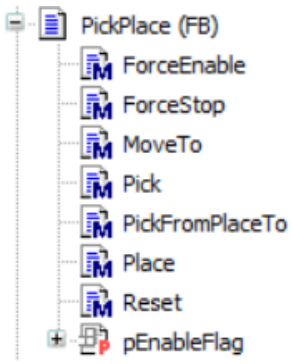



Figure 5.4: "PickPlace" class in CODESYS

### 5.2.3 The Conveyor Class

The "Conveyor" class was included in the class library to allow users to replicate the functionality of conveyors described in section 4.3.2.3. To adapt it to the library, two inputs were added: the "StartCondition" and "StopCondition", allowing users to easily enable and disable the conveyors without having to use the "ForceEnable()" and "ForceStop()" methods.

Figures 5.5 and 5.6 present the class diagram and the code tree obtained for this class.

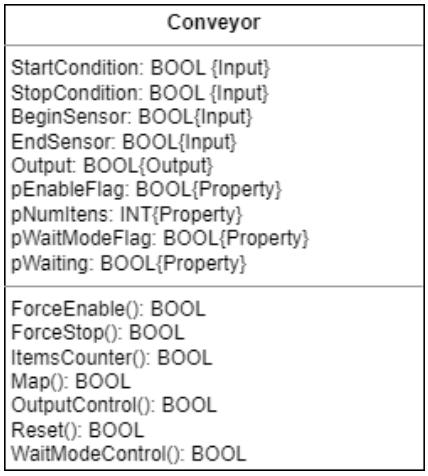



Figure 5.5: "Conveyor" class diagram

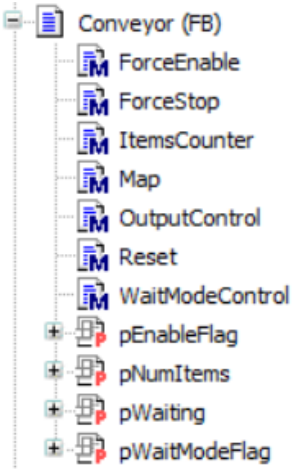



Figure 5.6: "Conveyor" class in CODESYS

### 5.2.4 The TurnTable Class

The "TurnTable" class was included in the class library to allow users to replicate the functionality of turntables described in section 4.3.2.4. To adapt it to the library, two inputs were added: the "StartCondition" and "StopCondition", allowing users to easily enable and disable the conveyors without having to use the "ForceEnable()" and "ForceStop()" methods.

Figures 5.7 and 5.8 present the class diagram and the code tree obtained for this class.

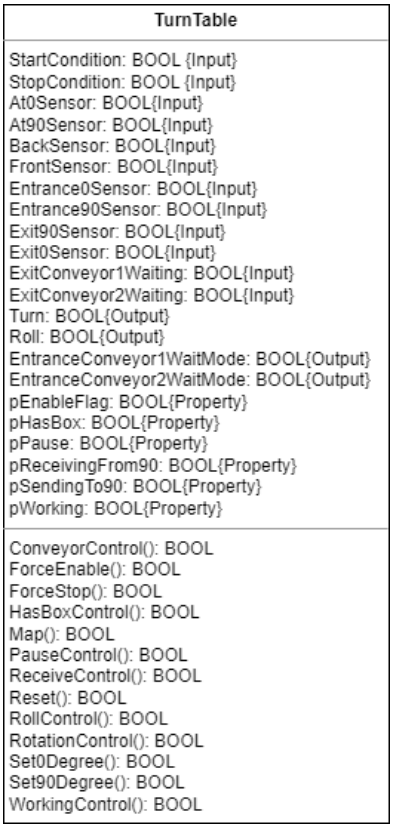



Figure 5.7: "TurnTable" class diagram

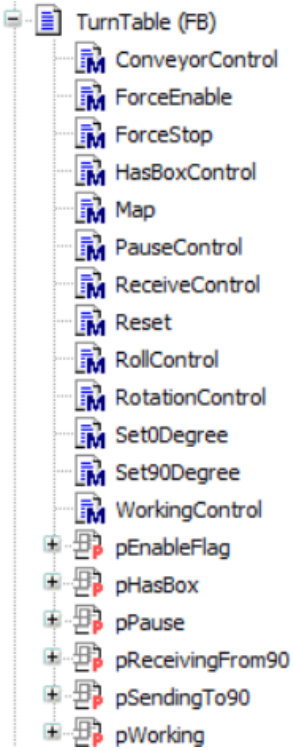



Figure 5.8: "TurnTable" class in CODESYS

### 5.2.5 The TwoAxisPickPlace Class

The "TwoAxisPickPlace" class included in the class library allows controlling, as the name suggests, the two-axis pick & place from Factory IO, presented in Figure 5.9, in the analog configuration. These elements can easily replace the cartesian pick & places used in the system of chapter 4, thus making the library more versatile.



Figure 5.9: Two axis pick & place

Its operation isn't very different from the "PickPlace" class defined in Section 4.3.2.2, however, there are some nuances that need to be taken into account.

Unlike the "PickPlace" class, which, when given the X, Y, and Z coordinates can automatically position the pick & place head at any point in space, in this class, there are only two axes, X and Z, with the rotation of the pick & place needing to be controlled separately through the "RotateCW" and "RotateCCW" outputs.

To accommodate this difference, taking the "PickPlace" class as a base, the following changes were made:

- The "Point" structure, originally designed for defining points in space with X, Y, and Z coordinates, was modified to accommodate points defined by X and Z coordinates along with the desired angle, being renamed to "PointXZAngle";
- A new method, "Rotate()", was added, which, receiving the desired angle as input (0, 90, 180, or -90), positions the pick & place correctly;
- A new property, "pCurrentAngle", was added to keep track of the current angle of the pick & place by monitoring the "RotateCW" and "RotateCCW" outputs;
- The "MoveTo()" and "PickFromPlaceTo()" methods were adjusted in order to function correctly with the two-axis pick & places.

Figures 5.10 and 5.11 present the class diagram and the code tree obtained for this class.

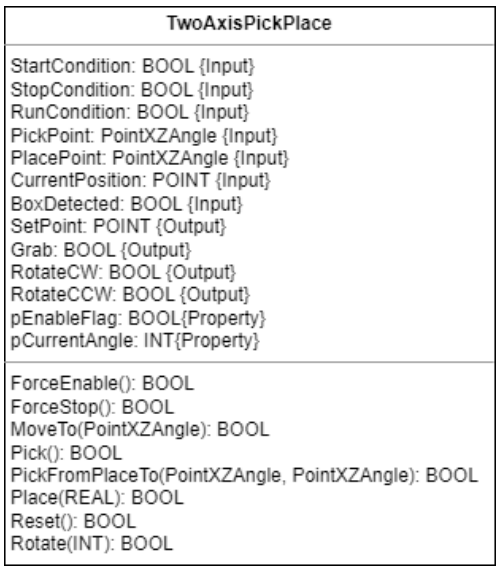



Figure 5.10: "TwoAxisPickPlace" class diagram

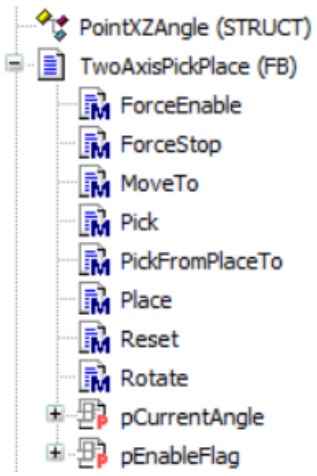



Figure 5.11: "TwoAxisPickPlace" class in CODESYS

In order to adapt the class to the library, similar to what was done for the "PickPlace" class, the following inputs were added to enable the control of the pick & places without the need for writing additional code:

- "StartCondition" and "StopCondition" - Boolean inputs that allow users to enable and disable the pick & place without having to use the "ForceEnable()" and "ForceStop()" methods;
- "PickPoint" and "PlacePoint" - "PointXZAngle" type inputs that allow users to define the pick location and the place location of the pick and place operation that can be initiated through the "RunCondition" input;
- "RunCondition" - Boolean input that allows users to initiate a pick and place operation from the "PickPoint" input to the "PlacePoint" input without having to use the "PickFromPlaceTo()" method.

### 5.2.6 The BidirectionalConveyor Class

The "BidirectionalConveyor" class included in the class library allows controlling, as the name suggests, bidirectional conveyors. These components can easily replace the conveyors

utilized in the system of chapter 4 enabling the addition of new features, thus making the library more versatile.

This class functions very similarly to the "Conveyor" class defined in section 4.3.2.3, with the only difference being the ability for the conveyors to operate in both directions. As such, to leverage the base operation defined in the "Conveyor" class, inheritance can be used with the following modifications:

- A new output, "OutputBackward", was added to allow the backwards operation of the conveyor;
- A new property, "pBackwardsMode", was added to allow the user to define whether the conveyor operates in forward or backward mode;
- The methods "ItemsCounter()", "Map()", "OutputControl()", "Reset()" and "WaitModeControl()" were overridden in order to take into account the direction of operation.

Figures 5.12 and 5.13 present the class diagram and the code tree obtained for this class.

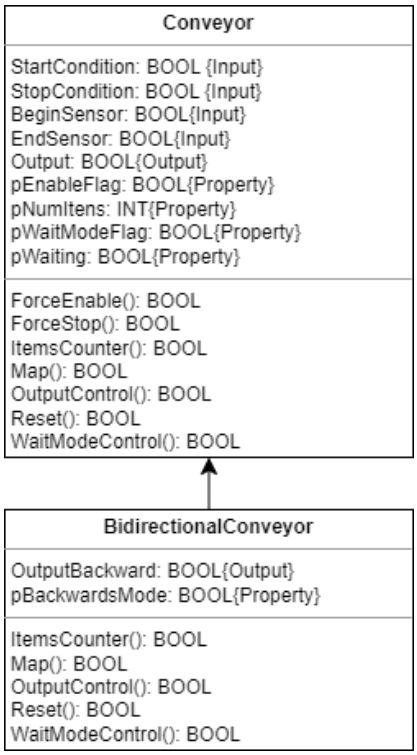



Figure 5.12: "BidirectionalConveyor" class diagram

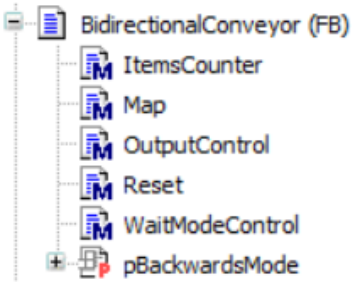



Figure 5.13: "BidirectionalConveyor" class in CODESYS

In order to adapt the class to the library, similar to what was done for the "Conveyor" class, two additional inputs were added: the "StartCondition" and "StopCondition", allowing users to easily enable and disable the conveyors without having to use the "ForceEnable()" and "ForceStop()" methods.

### 5.2.7 The ChainTransfer Class

The "ChainTransfer" class included in the class library allows controlling, as the name suggests, the chain transfer table from Factory IO presented in Figure 5.9. These elements function similarly to a conveyor but are capable of transporting items in two perpendicular directions. As such, they can easily replace the turntables utilized in the system of chapter 4, enhancing the versatility of the library.

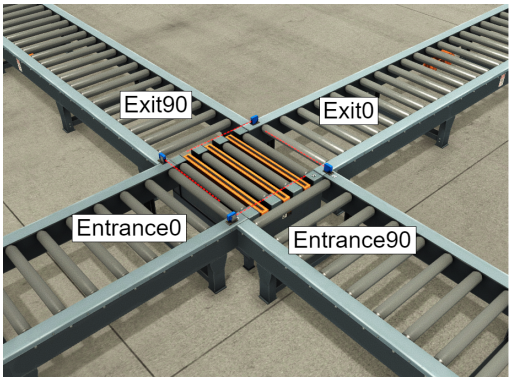



Figure 5.14: Chain transfer table

The operation of this class isn't very different from the "TurnTable" class defined in section 4.3.2.4, however, unlike turntables, chain transfer tables do not need to position themselves to receive items, as they can roll in both directions.

To accommodate this difference, taking the "TurnTable" class as a base, the following changes were made:

- Since chain transfers do not require rotation, the "RollControl()" and "RotationControl()" methods from the "TurnTable" class were modified to a single method responsible for controlling item transfer, named "SendControl()";
- Given the absence of rotation in chain transfers, the methods "Set0Degree()" and "Set90Degree()" were removed;
- The methods "ConveyorControl()" and "HasBoxControl()" were also adjusted to function correctly with chain transfers.

Figures 5.15 and 5.16 present the class diagram and the code tree obtained for this class.

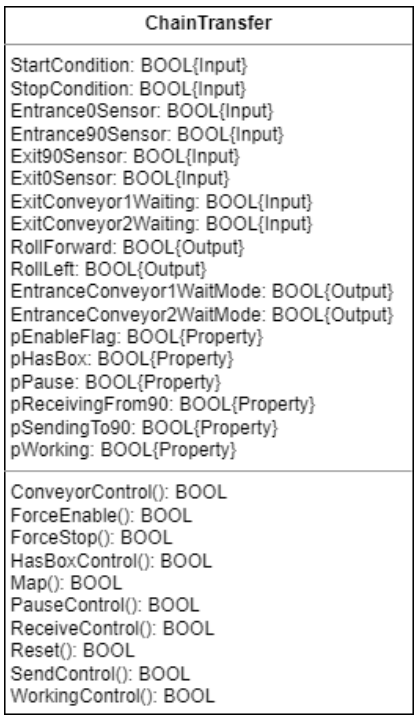



Figure 5.15: "ChainTransfer" class diagram

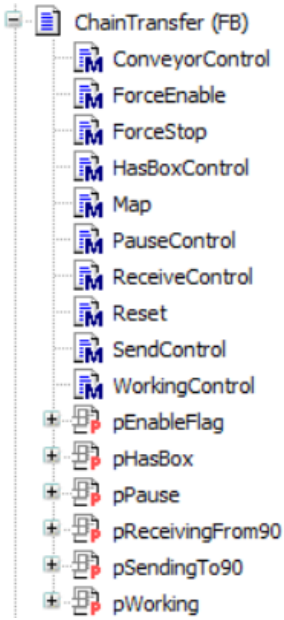



Figure 5.16: "ChainTransfer" class in CODESYS

In order to adapt the class to the library, similar to what was done for the "Conveyor" and "BidirectionalConveyor" classes, two additional inputs were added: the "StartCondition" and "StopCondition", allowing users to easily enable and disable the chain transfer tables without having to use the "ForceEnable()" and "ForceStop()" methods.

## 5.3 Class library application

To demonstrate the applicability of the developed class library and how it facilitates the development of OOP systems, this section will present an example of a plant PLC-controlled solution developed using the library.

### 5.3.1 The target workstation

The developed workstation, presented on Figure 5.17, aims to separate parts by weight, utilizing various conveyors, a scale conveyor, and a two-axis pick & place.

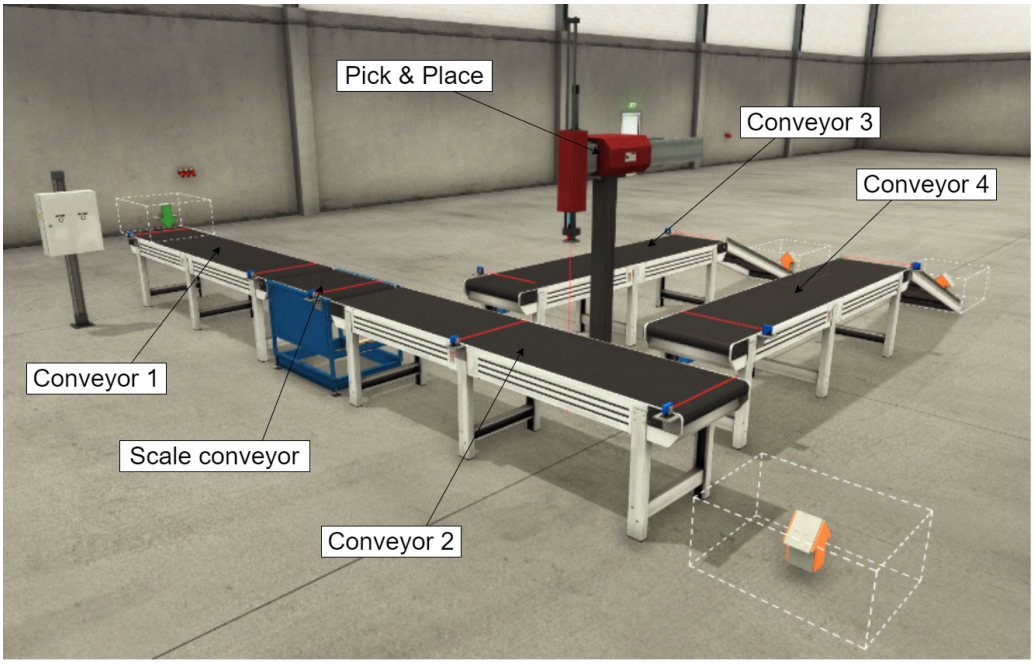



Figure 5.17: Sorting by weight workstation

Parts of different weights are manually placed at the beginning of conveyor 1 and transported towards the scale conveyor. Only one part enters the scale conveyor at a time, with the conveyor stopping for 1 second to weigh it. After the weighing process, the part moves to conveyor 2 where, depending on its weight, it is either picked up by the pick & place or continues until it is dropped off at the end of conveyor 2. In this case the workstation was programmed so that parts weighing less than 6kg are dropped at the end of conveyor 2, parts weighing between 6kg and 8.5kg are transported by the pick & place to conveyor 4, and parts weighing 8.5kg or more are transported by the pick & place to conveyor 3.

### 5.3.2 UML Modelling of the solution

Before diving into the actual programming, it is necessary to first consider how to approach the problem.

When it comes to the objects needed to control the workstation, it is clear that 4 conveyors, a scale conveyor, a pick & place and a queue for the weights of the parts are required. Considering

these objects, Figure 5.18 presents the sequence diagram of the intended behaviour for the workstation, considering that a part weighting between 6kg and 8.5kg is placed at the beginning of conveyor 1.

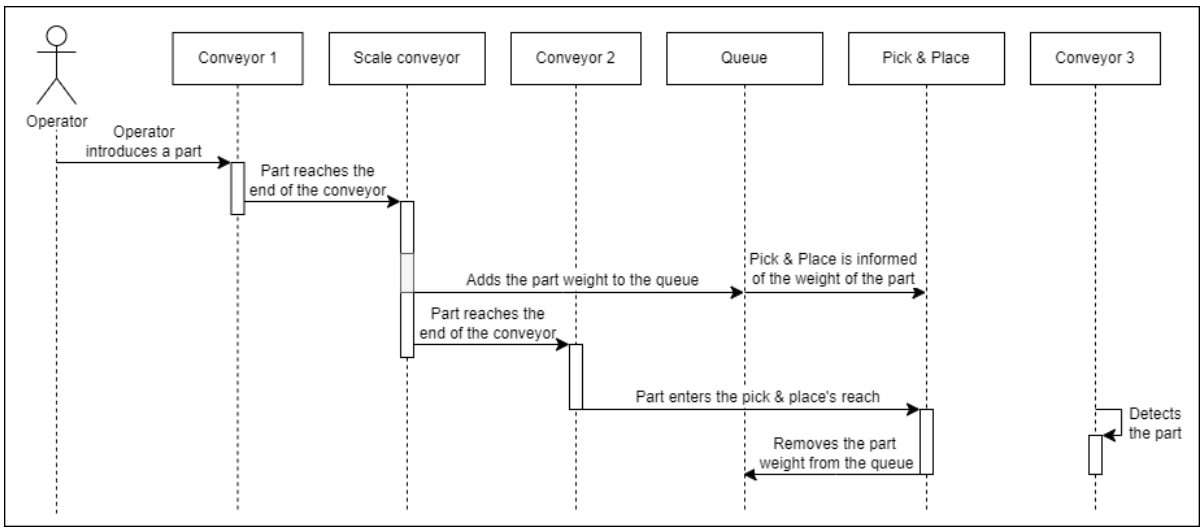



Figure 5.18: Sorting by weight sequence diagram

With the workstation behaviour defined, programming can now proceed.

### 5.3.3 PLC programming

Using the developed class library, programming the workstation was very straightforward, as practically all the necessary classes were readily available, reducing the amount of new code that needed to be developed.

To program the workstation, the following classes available in the library were utilized:

- "Queue" class - Responsible for controlling the queue with the weights of the parts on conveyor 2;
- "Conveyor" class - Responsible for controlling the conveyors;
- "TwoAxisPickPlace" class - Responsible for controlling the pick & place

In addition to the classes available in the library, it was necessary to create a new one: the "ScaleConveyor" class, responsible for controlling the scale conveyor. For that, and leveraging the flexibility of OOP through inheritance, the "ScaleConveyor" class was created by extending the "Conveyor" class defined in section 5.2.3, thereby inheriting all of its methods, properties, inputs, and outputs.

With the basic operation of the scale conveyor automatically defined, inherited from the "Conveyor" class, it was only necessary to specify its unique functionality. To achieve this:

- The input "ScaleValue" was added, which receives the voltage value corresponding to the weight of the part on the scale conveyor;
- The method "Measure()" was added, which allows stopping the conveyor for 1 second after the entry of a part to properly weight it;
- The property "pLastMeasure" was added, which allows to store the result of the last weighing so that it can be inserted into the queue.

Figures 5.19 and 5.20 present the class diagram and the code tree obtained for this class.

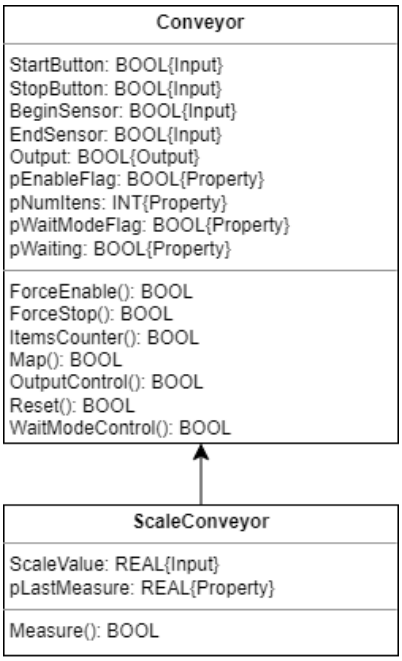



Figure 5.19: "ScaleConveyor" class diagram

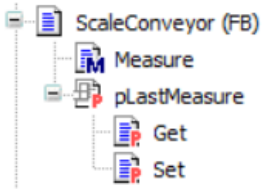



Figure 5.20: "ScaleConveyor" class in CODESYS

In this way, even though a class that was not initially defined in the library was necessary, it was possible to utilize a class from the library as a parent class, greatly facilitating the development of the new class.

With all classes defined, the "ScaleConveyor" class within the program itself and the other ones in the class library, it was only necessary to instantiate them in the PLC_PRG and manage the interactions between the objects.

## 5.4 Chapter summary

In this chapter, the concept of library was introduced, presenting its advantages and the most important aspects to consider during its development.

As seen, OOP libraries consist of collections of preprogrammed classes that programmers can use in their programs. Through the use of libraries, it is possible to streamline the development process of programs, preventing programmers from having to start from scratch and serving as a foundation where they can access proven code for common situations.

In order to develop a library, it is important to ensure that users understand how it works in order to utilize it effectively. As such, some important aspects were presented for library development, one of the main ones being the development of documentation to provide users with proper help and support when using it.

Finally, the class library developed within the scope of this document was presented. To demonstrate the advantages mentioned throughout the chapter, the class library was designed to allow users to replicate the system presented in chapter 4. Additionally, new classes were included to make the library more versatile, enabling users to create more varied systems and even modify the system presented in chapter 4.

All classes included in the developed library were described in this chapter, additionally, all the information needed to use each class was thoroughly detailed in the respective CODESYS documentation.

To demonstrate the utility of the developed class library, an application example of a workstation programmed using the available classes was presented. Besides exploring how to implement the library and showing how much it facilitates programming, this example also highlighted the possibility of creating new classes through inheritance using the ones available in the class library, further emphasizing how libraries can be useful.

The next chapter states the final conclusions of the dissertation and provides suggestions for future works.

# 6 Conclusion and Future work

The present dissertation, supported by the work of Eduardo Silva [4] and José Marín Sánchez [27], allowed for a discussion on object-oriented programming as a new paradigm for PLC programming, its advantages over traditional procedural programming, and how it may impact industrial systems programming.

After comparing object-oriented programming with procedural programming, it became evident that, with the increasing complexity of systems, more demanding requirements, and the need for flexibility, OOP becomes the obvious choice. By dividing the system into self-contained objects, OOP improves organization, facilitates interpretation, and through modularity, allows for system alterations and expansions with ease, making it the optimal choice for handling complex systems.

This programming paradigm was originally applied to information technology, thus prompting the question of whether operational technology can benefit from it. Through the comparison conducted in this dissertation, it was possible to conclude that, although information systems and automation systems have entirely different objectives, both utilize objects that, despite being distinct, lay on similar functional patterns and information exchange mechanisms, essentially serving the same purpose. Therefore, it was concluded that automation systems may not only draw insights from the way IT systems are engineered but also from the modeling tools they use.

By leveraging modularity, a major pillar of OOP, coupled with the fact that the objects comprising the system functioning almost like individual programs, the concept of OOP libraries emerged, enabling the usage of "technological objects" and "information objects" in other scenarios than those for which they were originally created for, while maintaining performance and abstraction. Putting this idea into practice, the last part of the document discussed the development of a library containing some classes that allow users to program Factory IO scenarios in a software approach that reassembles LEGO constructions. While the developed class library is by no means a complete tool, it serves as a model showcasing library development and utilization, which can be used to create simple systems or as a "CookBook" for other functions that those interested in the subject may want to add to the library.

Despite all the advantages presented regarding OOP, it is also important to understand that this programming paradigm is not a silver bullet nor a master key to solving any complex problem. OOP requires the support of complex modeling techniques, deep knowledge of a high-level programming language, and experienced designers and programmers. For simple cases, it is not the best solution, which means that in simple applications, state diagrams-based modeling and classic programming languages like ladder still prevail. Additionally, OOP being a relatively new concept in the industry has led to resistance among programmers, expected to diminish as technicians receive proper education, training, and witness positive outcomes firsthand.

In summary, the major conclusion of this dissertation is that object-oriented PLC programming is a modern tendency that addresses many issues encountered in procedural programming, not replacing it but serving as an alternative for complex systems requiring high flexibility. Furthermore, the use of OOP not only simplifies the development of complex systems and enhances comprehension but also facilitates the sharing of code among users via libraries. This fosters a culture of collaborative code-sharing, enriching the programming experience and enabling the creation of repositories housing solutions to a multitude of common problems.

Another important conclusion is that although derived from information systems, OOP in PLCs faces big differences: information systems are supported by complex multi-tread operating systems, are event-triggered, and adopt synchronous programming; PLCs have a simple

software model, are real-time, time-triggered systems and run applications asynchronously. Therefore, PLCOOP requires effort from those who are familiar with information technology.

## 6.1 Future work

The present dissertation serves solely as an introduction to the vast topic of object-oriented programming, and as such, there are several possibilities to expand the developed work by exploring various aspects that were not addressed.

One suggestion could be addressing the utilization OOP to develop HMIs, enabling the definition of HMI elements as objects, a topic that wasn't addressed in this document.

Furthermore, the developed class library could be expanded by adding more classes to provide users with greater flexibility, potentially including classes responsible for managing HMIs.

It would also be interesting to explore the use of RFIDs in object-oriented automation systems, as they enable more specific interactions between objects by allowing information to be associated with items in the system, thereby making interactions dependent on the item in question.

Another relevant subject that was not discussed is the research of methodologies for translating UML diagrams into executable code, similar to the established techniques used for converting from Grafcet.

Object-oriented design patters could also be explored in order to define general reusable solutions to common problems encountered in software design.

Lastly, it would also be relevant to explore security in OOP, which could involve, for example, exploring access specifiers to define the accessibility of classes, methods, and properties.

## 6.2 Personal assessment

Regarding the author's personal experience, developing this dissertation proved to be a valuable challenge. Through this project, the author was able to learn about a new topic he considers highly relevant today: object-oriented programming. This topic is still relatively recent, at least in the industry, but it shows great potential and could become the standard for PLC programming in large systems, making this dissertation highly valuable. This work also allowed the author to deepen knowledge acquired in other courses and to further develop his programming skills.

Throughout the dissertation, several difficulties had to be overcome. Initially, there was the challenge of becoming familiar with the concepts of object-oriented programming. This included a paradigm shift regarding UML, which is typically associated with information systems, and adapting to a new way of thinking when using OOP. However, these challenges also contributed to the dissertation's value. Overcoming them allowed the author to deepen his understanding of object-oriented programming and enhance his problem-solving and programming skills, ultimately making this work a significant personal and academic achievement.

## References


[1] Jeff Payne. "Back to the future of the PLC". *Control Engineering Magazine*, May 2024. Available: https://www.controleng.com/articles/back-to-the-future-of-the-plc/.

[2] International Electrotechnical Commission. "IEC 61131 Programmable controllers - Part 3: Programming Languages", 3rd Edition, 2013.

[3] Gary L. Pratt. "Leverage object-oriented industrial programming". *Control Engineering Magazine*, April 2019. Available: https://www.controleng.com/articles/leverage-object-oriented-industrial-programming/.

[4] Eduardo M. M. Silva. "Object-Oriented PLC Programming". Master's thesis, In Mechanical Engineering, Automation Branch. Faculdade de Engenharia da Universidade do Porto, 2020. Available: https://repositorio-aberto.up.pt/bitstream/10216/126744/2/390788.pdf.

[5] Visual Paradigm. "What is Unified Modeling Language (UML)?", Visual Paradigm Learning Guide. Available: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/.

[6] CODESYS Gmb H. "CODESYS Development System" The IEC 61131-3 programming tool for industrial control and automation technology. Internet Home Page: https://www.codesys.com/products/codesys-engineering/development-system.html.

[7] Real Games. Factory IO, Next-Gen PLC Training. Internet Home Page: https://factoryio.com/.

[8] AMCI - Advanced Micro Controls Inc. "What is a PLC?", Advanced Micro Controls, Tech Tutorials, 1999. Available: https://www.amci.com/industrial-automation-resources/plc-automation-tutorials/what-plc/.

[9] Vaidyanath Nanjundaiah. "PLC Ladder Logic Basics", EZAutomation. Available: https://www.ezautomation.net/industry-articles/plc-ladder-logic-basics.htm.

[10] John A. Stankovic. "Real-Time and Embedded Systems", 1996. Available: https://scholarworks.umass.edu/cs_faculty_pubs/142/.

[11] Mason White. *Mastering PLC Programming: The software engineering survival guide to automation programming.* Packt Publishing, 2023.

[12] Dr. Ken Ryan. "Coder's Corner: The IEC 61131-3 Software Model", Automation.com, 2003. Available: https://www.automation.com/en-us/articles/2003-1/coder-s-corner-the-iec-61131-3-software-model.

[13] PLCopen. "IEC 61131-3: a standard programming resource". Available: https://www.plcopen.org/.

[14] K. Eliason. "Difference Between Object-oriented Programming and Procedural Programming Languages", NeONBRAND Newsletter 2013. Available: https://neonbrand.com/websites/development/procedural-programming-vs-object-oriented-programming-a-review/.

[15] PLCopen - Promotional Committee 2. "Guidelines for usage of Object-Oriented Programming", Version 1.0 - Official Release, 2021. Available: https://plcopen.org/system/files/downloads/plcopen_oop_guidelines_version_1.0.pdf.

[16] Alexander S. Gillis. "Definition object-oriented programming (OOP)", TechTarget, 2021. Available: https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP.

[17] Erin Doherty. "What is object-oriented programming? OOP explained in depth", Educative, 2024. Available: https://www.educative.io/blog/object-oriented-programming.

[18] Alan Kay. "Alan Kay On Messaging", 1998. Available: https://wiki.c2.com/?AlanKayOnMessaging.

[19] Anjana Vakil. "Object Oriented Programming is not what I thought - Talk by Anjana Vakil", 2020. Video presentation. Available: https://www.youtube.com/watch?v=TbP2B1ijWr8.

[20] Raul Sidnei Wazlawick. *Object-Oriented Analysis and Design for Information Systems*. Morgan Kaufmann Publishers, 2014. Available: https://dahlan.unimal.ac.id/files/ebooks2/2014%20Object-Oriented%20Analysis%20and%20Design%20for%20Information%20Systems.pdf.

[21] Visual Paradigm. "What is SysML", Visual Paradigm Learning Guide, 2023. Available: https://guides.visual-paradigm.com/what-is-sysml/.

[22] Marshall Gunnell. "Operational Technology (OT)", Techopedia, 2023. Available: https://www.techopedia.com/definition/operational-technology-ot.

[23] Vladimir Zwass. "Information system", Encyclopedia Britannica, 2024. Available: https://www.britannica.com/topic/information-system.

[24] Trend Micro. "Industrial Control System". Available: https://www.trendmicro.com/vinfo/us/security/definition/industrial-control-system.

[25] Ramotion. "Synchronous vs Asynchronous Programming: Models, Differences, Use Cases", 2023. Available: https://www.ramotion.com/blog/synchronous-vs-asynchronous-programming/.

[26] MathWorks. "What Is Reentrant Code?", MathWorks documentation. Available: https://www.mathworks.com/help/ecoder/ug/what-is-reentrant-code-2d70b58a9a46.html.

[27] José Marín Sánchez. "Diseño de un sistema de control distribuido usando Factory IO y Codesys V3 (comunication through OPC)". Master's thesis, Escuela Técnica Superior de Ingeniería Universidad de Sevilla, 2018. Available: https://idus.us.es/bitstream/handle/11441/83982/TFG-1778-MARIN.pdf?sequence=1&isAllowed=y.

[28] CODESYS. CODESYS Online Help. Available: https://help.codesys.com/.

All web pages were available at the dissertation's submission date (26/07/2024).