Object-Oriented PLC Programming

Eduardo Miranda Moreira da Silva

Master's Dissertation

Supervisor: Prof. António José Pessoa de Magalhães

Master's Degree in Mechanical Engineering
Automation Branch

January 27, 2020

Abstract

This document aims to investigate how Object-Oriented Programming (OOP) can improve Programmable Logic Controllers (PLC) programming. To achieve this, a PLC project was built using the OOP approaches suggested by the International Electrotechnical Commission (IEC) 61131-3 Standard. This project was tested on a simple but realistic simulated scenario for evaluation purposes.

The text starts by exposing the history of PLC programming, it's recent enhancements and the rise of object-oriented programming in the industry and how it compares to regular software programming, before briefly presenting the resources that support object-oriented PLC programming. Four case studies and their controlling applications are then introduced, along with examples of OOP usage. The dissertation ends with a comparison between applications designed with and without using OOP.

OOP allows the creation of a standard framework for similar groups of components, reduction of code complexity and easier and safer data management. Therefore, the result of the project was an easily customizable case scenario with "plug & play" components.

In the future, the idea is to build an HMI that can take care of the changes applied in the physical system (e.g., switching a component) without accessing the code.

Keywords: Industrial Software Development, PLC Programming, IEC 61131-3 Standard, Object-Oriented Programming.

Resumo

Este documento tem como objetivo investigar até que ponto a Programação Orientada a Objetos pode melhorar a Programação de PLCs. Para atingir esse objetivo, foi desenvolvido um projeto de PLC utilizando as abordagens de programação orientada a objetos sugeridas pela norma 61131-3 da *International Electrotechnical Commission*. Este projeto foi testado num ambiente simulado mas realista para fins de validação.

O texto começa por expor o histórico da programação de PLC, as suas recentes melhorias e o aparecimento da programação orientada a objetos na indústria, juntamente com uma comparação com a programação de software comum, antes de apresentar os recursos que suportam a programação de PLC orientada a objetos. Quatro casos de estudo e a aplicação que os controla são, depois, apresentados, juntamente com exemplos de uso da programação orientada a objetos. A dissertação termina com uma comparação entre aplicações realizadas com e sem o uso da programação orientada a objetos.

A programação orientada a objetos permite a criação de uma estrutura padrão para grupos de componentes semelhantes, a redução da complexidade do código e a gestão de dados torna-se mais fácil e segura. Portanto, o resultado do projeto foi uma aplicação facilmente personalizável com componentes "*plug & play*".

No futuro, a ideia é construir uma HMI que possa suportar alterações no sistema físico (por exemplo, alternar um componente) sem necessidade de alterações no código.

Palavras-chave: Desenvolvimento de Software Industrial, Programação de PLCs, Norma IEC 61131-3, Programação Orientada a Objetos.

Acknowledgements

I'd like to thank Professor António José Pessoa de Magalhães for his guidence and support throughout the execution of this document, especially for provinding most of the resources that I used.

I'd also like to thank my mother for helping me, even if not directly, as her support was definitely really important for me to be able to complete this dissertation.

Table of Contents

Acronyms

FB – Function Block;
GRAFCET – *GRAphe Fonctionnel de Commande, Étapes Transitions*;
IEC – International Electrotechnical Commission;
I/O – Inputs/Outputs;
OOP – Object Oriented Programming;
PLC – Programmable Logic Controller;
POU – Program Organization Unit;
UDT – User-Defined Types;
UML – Unified Modeling Language.

Table of Figures

Table of Tables

1 Introduction

As the PLC industry evolves and requirements become increasingly sophisticated, the use of more powerful PLC programming approaches naturally turns into a need. What once was treated as powerful but very specific and hard to learn resource is slowly gaining importance in the world of industrial control.

Programming is the bedrock of the age of information technologies. It is so powerful that it feels like a computer is capable of doing almost anything, a tool that provides endless possibilities, only limited by hardware resources and the skill of the programmer. So, PLC programmers started to think about ways of bringing proved powerful software tools and approaches into the industrial world.

The introduction of high-level programming approaches to the PLC industry has not only allowed programmers to come up with more powerful solutions but also get rid of some of the industry's biggest problems.

Therefore, the aim of this document is to analyze how Object-Oriented Programming can improve the industry, using it to create PLC applications and testing them on simulated scenarios.

1.1 Aims

The goal of this dissertation is to show how object-oriented programming can improve PLC programming. In order to reach that goal, the creation of PLC applications using OOP to control individual components is going to be analyzed in four case studies.

In order to analyze the impact of OOP's features, an existent system will be changed using this programming approach. Both systems will be compared in order to understand the how object-oriented programming is able to improve systems.

1.2 Research Methodology and Project Execution

Before addressing any case scenario, it is necessary to understand why object-oriented programming has been gaining importance in the PLC industry. Figure 1.1 shows the order of execution of the tasks that were performed in order to achieve the objectives exposed in the previous section.

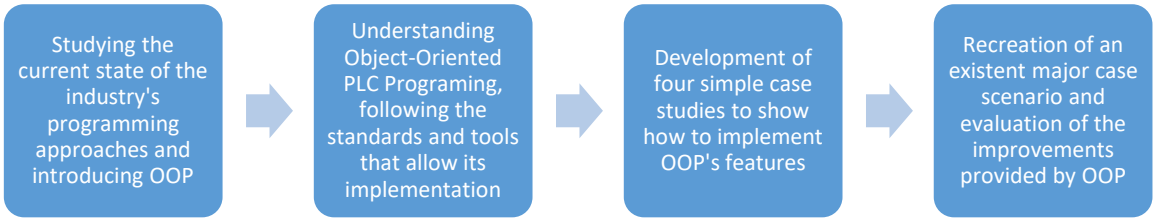


Figure 1.1 - Research Methodology and Project Execution

The first step is studying the current state of the industry's programming approaches and their limitations as well as how an approach that is normally used to design regular software such as object-oriented programming can deal with such limitations.

The second step is understanding how to program using object-oriented approaches, and which standards and tools can be used to design PLC applications using this type of approach. Understanding OOP's features and how they can be used to create powerful applications is essential to understand the development of the case studies.

The third step is the development of four simple case studies to demonstrate OOP's features can be used to control some individual components.

The fourth step is the recreation of a previously created major scenario using object-oriented programming. Both scenarios are compared in order to evaluate how OOP can improve a major automated system.

1.3 Dissertation's organization

In chapter 2, classical and current PLC programming approaches are introduced, as well as their limitations. Object-oriented programming is then introduced as an alternative to the most common programming approaches used in the PLC industry. It is also compared to regular object-oriented programming. The chapter ends explaining OOP's role in the PLC industry and how it can be used alongside the industry's common practices.

In chapter 3, the standards and tools that allow the creation of object-oriented PLC applications are introduced. In particular, the 3rd part of the IEC 61131 standard is introduced since it defines the programming languages for programmable controllers. Its most important features are thoroughly explained.

In chapter 4, four simple case studies are introduced in order to explain how OOP's features can be implemented to create powerful applications to control individual components.

In chapter 5, a previously designed major scenario was recreated using object-oriented programming. Both final results were compared in order to evaluate the improvements provided by OOP.

In chapter 6, the work that was carried out throughout the dissertation is presented while justifying the main conclusions and benefits of this project to the industry and to the author, as well as some instances of future work.

2 PLC Programming Engineering

PLCs and their associated programming practices have experienced great transformations since the 1960s. Understanding the evolution and history of PLCs and their programming methodologies, as well as how these compare to general software programming is essential for PLC based projects organization, execution and maintenance. Traditional programming methodologies are easy to learn and apply, but they are also inefficient and error prone. Yet, recent approaches brought a set of features to enhance PLC's controlling capabilities.

This chapter surveys the evolution of PLC programming over time and how classical programming compares to the most recent programming approaches, namely object-oriented programming which will be the research subject of this document.

2.1 PLCs: Emergence and Early Software Development

A PLC (Programmable Logic Controller) is an industrial computer adapted for the control of manufacturing processes [1]. PLCs have emerged as a replacement for hard-wired relay systems, because of their flexibility and ease of programming. Thus, they must be able to work in harsh usage environments such as strong vibrations or severe temperatures.

In the beginning, PLCs were programmed using Ladder Logic [2] which strongly resembles schematic diagrams of relay logic (Figure 2.1). A scanning engine and a memory management stack are always associated with ladder logic: at first, physical inputs are read and stored in an input memory table; then, the ladder logic is run, computing the output memory table; at the end of the logic cycle, the physical outputs are updated according to the output memory table.

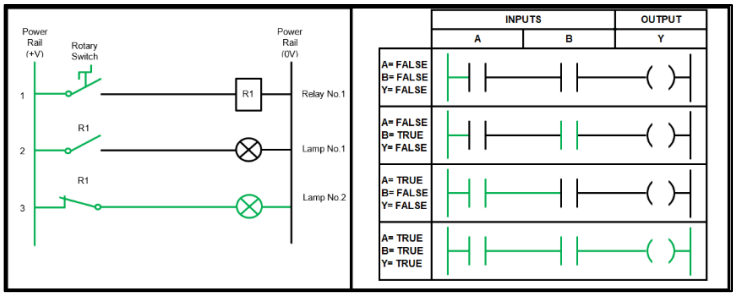


Figure 2.1 - Left: relay logic; right: ladder logic [3]

As a major benefit, this programming notation is easy to understand, learn, troubleshoot and debug, even by non-graduate automation technicians. Moreover, it does an excellent job at representing discrete logic: each line of code must be true to turn something on. Ladder logic is thus natural for machine and process control and allows for writing programs with a certain level of complexity. Moreover, the source code and descriptions of the programs are often stored in the controller, which allows maintenance personnel to easily access it in order to troubleshoot it [4].

However, ladder logic has some limitations, namely regarding data structures and protection. Ladder diagrams traditionally address memory in single bits or 16 bits registers and are allowed to read and write data to variables anywhere in the program. Data protection or the creation of a data structure is thus difficult in ladder, as accessing can be done freely and directly, making it easy to accidentally access or change the wrong data, thus causing unexpected behaviors that can lead to catastrophic failures. Finding a way to protect internal information is also a difficult job, as data can be corrupted by faulty code anywhere in the program. The use of named variables could be a solution, but even in this case, variables could allocate their values in overlapping memory locations. Modern editors minimize this kind of errors as they include tools that show which memory locations are being used for each variable, checking and warning about possible conflicts [5].

Ladder logic used to deal only with boolean algebra, counters/timers and simple integer arithmetic but nowadays it supports mathematical operations through the use of more advanced function blocks. However, the inputs and outputs of the function blocks that perform such operations are not casted together in a data structure but, instead, referenced to individual memory locations. As a result, simple math operations are still easily performed – e.g., a subtraction like the one shown in Figure 2.2 –, but complex algorithms involving lots of variables and intermediate results can be difficult to program, debug, edit and document.

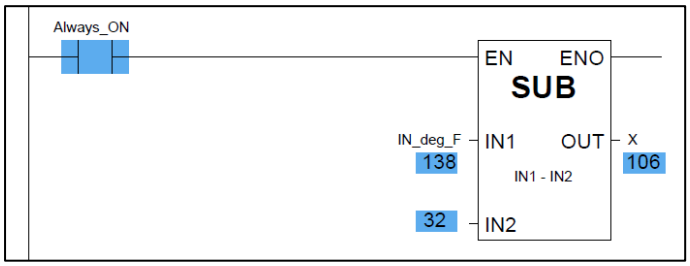


Figure 2.2 - A subtraction function block in a ladder diagram [6]

There are also some issues with limited execution control. Ladder diagram programs are executed in a left-to-right, top-to-bottom basis, and response time is defined by the speed at which the PLC can scan and execute. This works very well for many applications but not as much when the program execution needs to be flexible enough to adapt to operational mode changes of the controlled object. Jump commands can be used to transfer program execution to different parts of the diagram, but because overall response time depends on the length and complexity of a ladder program, it can become a problem when designing real-time systems – e.g., systems that need to respond within a limited time interval. Rearranging the structure of a program during runtime requires time and resources, and might be unacceptable for some cases like analog control (such as PID), which depends on properly managed memory values and on predictable and bounded execution timing.

Additionally, ladder programs that don't rely on a proper modular functional organization don't allow code to be reused. Unstructured programs can become very long and difficult to understand and manage. Even though simple programs are very easy to edit and debug, this task can become difficult if the program reaches larger sizes. Even though almost every major ladder diagram package includes functions and function blocks that can be called from the ladder rungs, many of them support limited numbers of subroutines or function blocks, making it still hard to break large programs into manageable parts [5].

As industrial control requirements became more and more challenging and sophisticated, new programming approaches started appearing. At first, PLCs could only read a very limited set of elementary data types (bits, bytes, etc.). Then came the introduction of analog inputs and outputs, and more recently PLCs started including a wider set of mathematical and programming functions that tend to be very difficult or even impossible to implement in traditional "ladder logic" or "function block" diagrams. On the other hand, PLC applications became strongly distributed and, consequently, modular programming and code replication techniques started playing a major role in PLC software engineering.

In short, PLC programmers and PLC manufacturers are constantly looking for ways to efficiently implement answers to the always emerging needs coming from the industrial world. Adopting object-oriented programming approaches in industrial programming is the most recent innovation [7].

2.2 Design Approaches for PLC Applications

As stated, programming concepts normally applied to regular computer programming recently started to be applied to PLCs in order to fulfil some of the needs that traditional ladder logic couldn't deal with. This section surveys this evolution.

2.2.1 Procedural Programming

At first, **procedural programming** was the way PLC programs were based on. This approach directly follows the **Top Down Design** method [8]. It takes on applications by solving problems (procedures) from the top to the bottom of the code. The program starts with a major procedure, breaking it down into properly ordered and detailed sub-procedures until it is simple enough to be solved [9].

A scan cycle of a PLC has 3 main steps: 1) input reading; 2) program execution; 3) output writing (as stated in section 2.1). One could say that this input to output data processing order closely implements a Top Down approach. As such, PLC's operating systems seem to naturally enforce procedural programming.

Whilst this is somewhat true, *ad-hoc* procedural programming tends to be **hard to edit and manage**; so, even though procedural programming introduced lots of new functionalities, some of the biggest advantages that defined PLC programming were lost – i.e. the ease of programming and troubleshooting. Also, some other software problems, such as multitasking support and openness to high level programming started to become an issue when designing PLC applications.

Procedural programming was made very useful at first because of the introduction of custom functions; custom blocks could be used to perform self-contained or complex jobs on volatile data that were difficult to program using conventional PLC programming languages. Yet, functions – later evolved to static data functions of function blocks – made it possible to use persistent data blocks in ladder diagrams and other emerging PLC programming languages. This solution, often described as "almost object-oriented" was very successful and is widely used nowadays.

Since this solution fit PLC programming so well, programmers kept on exploring how other object-oriented concepts could be applied onto PLC programming in order to deal with procedural programming's disadvantages.

2.2.2 Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming model based on the concept of **objects** [9]. Using this approach, programs are built by objects that interact with each other. Objects can contain data in the form of **properties** (also known as fields or attributes) and code in the form of **methods** (or procedures). It focuses on the **data to be manipulated** rather than the logic to manipulate it.

In software engineering, just like in real life, an **object** can be anything we see - like a pen or a computer; Yet, in the second context, an object can also be an imaterial entity - like a bank account. Objects have different but typical properties and behaviors that differentiate from each other. For example, a pen has a set of properties (such as body color, writing color, brand) and can perform certain tasks (like cap, uncap and write) called behaviors or operations. Different objects have different sets of properties and behaviors, while similar objects (i.e., objects from the same class) exhibit the same sets of properties and behaviors.

In software engineering, an object is a closed group of data and code, whose execution may require external input data or make internal data available for external use, but it can only access the data that it needs to run. This means that an object will only use the data that it is supposed to manipulate, none of the redundant data to a certain action will be solicitated. This improves system security and avoids data corruption. The data inside a certain object can also be **hidden** from other objects, which means that external access to this data is denied.

Object-Oriented Programming languages can be very different from each other, as they may have different purposes. So, it is worth noting that, in this document, class-based object-oriented programming concepts will be studied in the context of their applicability in PLC controlled systems.

An **object** is thus an instance of a **class**. To explain the concept of a class, Figure 2.3 introduces another real-life example: different cell phones can belong to different brands, and can have different colors, shapes and hardware specifications. A white iPhone and a black Samsung Galaxy are different objects, but both of them can be classified as cell phones.

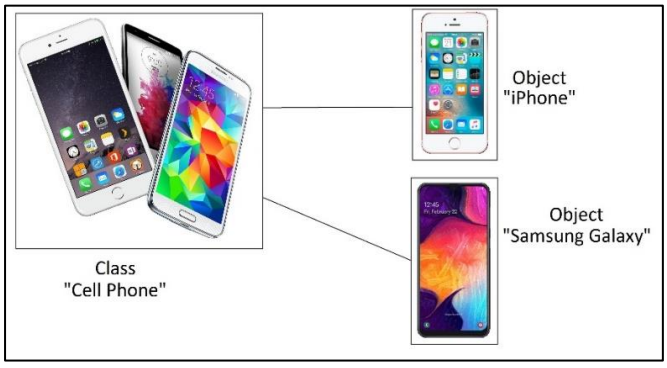


Figure 2.3 - "Cell Phone" class and some examples of objects

The **class** is just a conceptual framework that defines the type of properties and behaviors that a group of objects must have, whilst the **object** possesses said properties and behaviours.

An object-oriented program features multiple classes to create a standardized framework. The creation of a new object is, thus, easier and their interactions with other objects is easier to implement and can be easily managed.

2.2.3 Object-Oriented Programming: advantages over Procedural Programming

Object-oriented programming promotes **code reusability** and **ease of maintenance**. If a "phone" class was created at some point in an application, then it can just be called again in the same application. Code maintenance is also very easy to carry, as it is possible to change the class's code therefore changing all the objects that implement it.

It is also very easy to create new similar objects by **extending** the existing ones. That extension could **add new features** or **override** existing functionalities. Changes in the base classes are applied to every class that extends it, thus eliminating the need for **copying and pasting**.

It's important to point out that, even though OOP enhances **simplicity** and the **capacity to implement future changes**, it doesn't necessarily make coding faster. Object-Oriented Programming's major benefits come from the following concepts:

- **Encapsulation**

Encapsulation is used to bundle data with the methods that operate on it and to hide data inside a class, preventing unauthorized parties to directly access it. It reduces code complexity and increases reusability. The separation the code allows the creation of routines that can be reused instead of copying and pasting code, reducing the complexity of the main program.

- **Abstraction**

Abstraction is the process of hiding important information, showing only the most essential information. It reduces code complexity and isolates the impact of changes.

Abstraction can be understood from a real-life example: turning on a television must only require clicking on a button, as people don't need to know or the process that it goes through.

Even though that process can be complex and important, there is no need for the user to know how it is implemented. The important information that **isn't required** is hidden from the user, reducing **code complexity**, enhancing **data hiding** and **reusability**, thus making function blocks easier to implement and modify.

- **Inheritance**

Inheritance allows the user to create classes based on other classes. The inherited classes can use the base class's functionalities as well as some additional functionalities that the user may define. It eliminates redundant code, prevents copying and pasting and makes expansion easier.

This is very useful because it allows classes to be **extended** of **modified** (overridden) without changing the base class's code implementation.

What do an **old landline phone** and a **smartphone** have in common? Both of them can be classified as **phones**. Should they be classified as objects? No, as they also define the properties and behaviors of a group of objects. A smartphone works just like a regular phone, but it is also able to take pictures, navigate the internet, and do many other things. So, **old landline phone** and **smartphone** are child classes that **extend** the parent **phone** class.

- **Polymorphism**

The concept of **polymorphism** is derived by the combination of two words: Poly (Many) and Morphism (Form). It refactors ugly and complex switch cases/case statements.

Polymorphism allows an object to change its appearance and performance depending on the practical situation in order to be able to carry out a particular task [10]. It can be either static or dynamic: static polymorphism occurs when the object's type is defined by the compiler; dynamic polymorphism occurs when the type is determined during run-time, making it possible for a same variable to access different objects while the program is running.

A good example to explain polymorphism is a Swiss Army Knife (Figure 2.4):

Figure 2.4 - Swiss Army Knife

A Swiss Army Knife is a single tool that includes a bunch of resources that can be used to solve different issues. Selecting the proper tool, a Swiss Army Knife can be used to efficiently perform a certain set of valuable tasks. In the dual way, a simple adder block that adapts itself to cope with, for instance, *int*, *float*, *string*, and *time* data types is an example of a polymorphic programming resource.

2.2.4 Object-Oriented PLC Programming vs Computer Programming

The creator of the "object-oriented" concept didn't, originally, mean to create C++ or any of the class based Object-Oriented Programming languages. However, Object-Oriented PLC Programming is highly based in C++ and Java's principles.

According to Alan Kay [11], the most essential concepts for OOP are **encapsulation**, **message passing** and **dynamic binding** (the ability for the program to evolve/adapt during runtime). Concepts like **Classes, inheritance, special treatment for functions or data, polymorphism**, etc. aren't necessary because his goal is to *"get rid of data", "[…] to not have to worry about what's inside of an object. Objects made on different machines and with different languages should be able to talk to each other […]"*.

However, data types are very important in PLC programming, as they need to be clearly defined, mainly when dealing with the input and output signals of the PLC. If the variable type isn't clearly or correctly defined, the external signals won't be converted accurately.

For an application to be programmed and downloaded to a PLC, special software must be used. The code of the application must be compiled before being downloaded to the PLC so that it works efficiently and in **real-time**.

Apart from variables, there isn't much in a common PLC controlled system that could be described as "dynamic" in the computing sense. Object-oriented PLC programming software doesn't usually allow the creation and elimination of objects during run-time, simply because it has no applicability in PLC controlled systems. For example, if a PLC controls a system that incorporates 3 conveyors, then there should only be 3 conveyor objects. The creation and elimination of extra conveyor objects or the elimination of existing ones isn't allowed (or

required) during run-time. If a conveyor is to be added to/removed from the physical system, then a cold change must be performed (a change that is made while the system is offline – requires a redownload onto the PLC).

On the other hand, if an object stops being used, then it is only occupying space in the memory of the program. If the program allows memory allocation (the creation and elimination of objects during runtime), **memory exhaustion** might be an issue. The use of an automatic **garbage collector** like the one that exists in Java, which frees unused memory spaces, is not recommended for PLC application design software as it uses additional resources and impacts performance, as it takes time to be performed, and can possibly stall program execution. Since PLCs control **real-time environments** and demand tasks to be completed within limited time, a garbage collector is highly inappropriate as it could lead to unexpected and prejudicial results. Therefore, traditional OOP issues related to dynamic object creation and elimination are not present in Object-Oriented PLC Programming.

Yet, PLC programs often require lots of changes. However, most changes must be made while the system is online, as every second counts in common PLC controlled environments. Performing new downloads and having to re-compile the code every time a change is done takes precious time that cannot be spared by working systems, especially in largest ones. PLC application design software must allow UDTs (User-defined types) to be used/changed during run-time in order for OOP to reach its full potential in the PLC industry.

2.3 Complementing Traditional PLC Programming Practices with Object-Oriented Approaches

Object-Oriented Programming is clearly more adapted to program complex PLC applications than procedural programming. It also brings a very powerful set of features and advantages for the system.

OOP's major advantages over common PLC programming approaches are:

- **Code encapsulation, resulting in portable and reusable code**

Object-oriented programmed applications are extremely portable and reusable. Since classes are created separately, a programmer can create custom libraries with classes that he wishes to reuse in other projects without having to implement the rest of the program or without changing the program where it was created in the first place.

- **Better data management**

Data management is no longer done directly, which greatly decreases error proneness. Each data piece has to be declared, typified and initialized. It also becomes much easier to implement complex mathematical or programming functions: mathematical operations and programming functions like loops become much easier to implement, edit and document. This type of task was difficult to program using ladder logic.

- **Creation of standardized frameworks, which makes implementation and modification easier**

Having standardized frameworks for the constituents of a system is very important, as it standardizes how its components interact with each other, thus easing their implementation.

- **Taught in nearly every computer programming**

Even though OOP isn't as intuitive or as easy to learn as "ladder logic", chances are that younger engineers or scientists will have some knowledge in this subject. OOP isn't a recent concept, it was introduced in the 1960s or even before, but only recently was it applied to PLCs

in general. There is a lot of documentation available online about OOP, if one is interested in learning how to use it.

- **Runs in various hardware platforms**

Software applications programmed using OOP can run in different hardware platforms, while "ladder logic" programmed software must be run using a specific hardware.

However, it still needs to be implemented using specific software. The source code must be compiled before it is downloaded to the controller and most times it isn't present in the processor memory, meaning that it should be backed up carefully because the compiled code is usually not editable. It can be difficult to perform monitoring in real time. Libraries need to be connected to other resources used during compilation. If these connections and resources aren't understood, it will be difficult to get the program to run.

Industrial Automation has relied on other common PLC programming languages (mainly ladder logic) for nearly 50 years, and senior PLC programmers take it as an interesting and easy to cope language.

Although not as powerful as OOP, its advantages regarding simplicity are a very solid reason for a programmer not wanting to change. OOP is, thus, a very powerful high-level programming language that can be used as a (complementary) alternative to other PLC programming approaches.

Common PLC programming languages (such as Ladder Logic or Function Block Diagram) keep some functional advantages over OOP:

- Lower overhead, since they tend to need less memory and processing power;
- Easier to troubleshoot for maintenance personnel;
- Predictable behavior and worst-case execution time;
- Must be compiled so source code can be uploaded to the processor.

All programming approaches will coexist in the industry for the decades to come, as OOP is slowly, but surely, gaining importance in the automation field.

2.4 Concluding Remarks

In this chapter, the most significant PLC programming approaches were reviewed to better understand their capabilities and limitations. Ladder diagram's evolution has been able to cope with most of the needs that emerged over time. Its simplicity and ease of programming and maintenance have kept it as the most reliable programming approach over the years.

As data management became more and more important and programming needs became more sophisticated, other programming approaches started to be explored. However, the advantages that regarded simplicity, ease of programming and troubleshooting were lost.

Object-Oriented Programming doesn't bring back every advantage that made ladder logic and other traditional PLC programming languages great, as it is still requires high-level programming knowledge, but its advantages regarding data management, code reusability and overall programming capabilities make it a very important tool for the present and future of PLC programming.

The next chapter addresses the current tools and standards that support the implementation of OOP in PLC programming.

3 Standards and Tools for Object-Oriented PLC Programming

The existence of standards and tools that support Object-Oriented PLC Programming is essential to disseminate related concepts and practices. The standardization of PLC programming approaches and their implementation in practical resources has contributed to a more widespread use and approximation between different manufacturers' software.

This chapter addresses the standard that currently deals with PLC programming methodologies and some of the tools that implement the concepts it suggests.

3.1 The IEC 61131-3 Standard

The first edition of the third part (out of ten) of the IEC 61131 international standard was published in December 1993 by the International Electrotechnical Commission (IEC). It became known as the IEC 1131-3 standard before a later major change in IEC standards' numbering system and as IEC 61131-3 thereafter [12]. This part of the standard deals with PLC programming languages and software and PLC program execution models.

The first edition's basic principle is that a programmer can develop a control algorithm using any combination of the following five – three graphical and two textual – programming languages:

- Ladder Diagram (LD) – Graphical;
- Function Block Diagram (FBD) – Graphical;
- Structured Text (ST) – Textual;
- Instruction List (IL) – Textual;
- Sequential Function Chart (SFC) – Graphical.

Regardless of the used programming language, a control algorithm includes entities referred to as Program Organization Units (POUs). These can be reused within an application and can also be organized in user-defined libraries for importation into other control programs.

When creating a POU, the user can choose one of the following options:

- **Function** - this type of POU typically includes some code and some volatile data; it isn't instantiated in the main program. The user can call a function whenever needed. It only returns one output. A function can either perform a standard set of instructions or a user defined one;
- **Function Block or Class** - this type of POU includes some code and some non-volatile data; function blocks are instantiated and can return multiple outputs. A function block can also perform standard sets of instructions or user defined ones;
- **Programs**, such as the main program, are POUs that can be put to run or to stop. They are created in any of the above-mentioned standard languages, and can incorporate unique code and functions or function blocks previously created within a project or

referenced to external libraries. A program is the only POU type that can be inserted into a task.

All variables within a program must be declared, either locally to a POU or globally to the project, regardless of the POU or language used.

The goal of the standard is to be a **guide for PLC programming** and **not a strict set of rules**, it is expected that the controllers are only partially compliant due to the high amount of detail. The implementation of the standard's concepts allows more powerful communications between controllers produced by different manufacturers.

Meanwhile, two major revisions were conducted on the original IEC 61131-3 standard. The third and current edition of the standard was published in February 2013 [13].

The IEC 61131-3 standard deals with basic software architecture and programming languages of the control program within PLC since its first edition, but it now explicitly includes references to Object-Oriented Programming.

The latest edition of the IEC 61131-3 introduces the following Object-Oriented Programming features:

- Classes;
- Methods;
- Properties;
- Inheritance (along with access specifiers and polymorphism);
- Interfaces and Abstraction (polymorphism is also included here).

3.1.1 Classes and Function Blocks

According to IEC 61131-3, a **class** is a POU designed for object-oriented programming. It contains essentially variables and methods. A class shall be instantiated before its methods can be called or its variables can be accessed.

A class differs from a function block in the aspects described in Table 3.1:

Table 3.1 - Differences between a class and a function block

	Function Block	Class
Keywords (to create the POU)	FUNCTION_BLOCK END_FUNCTION_BLOCK	CLASS END_CLASS
Variable declaration	Only in the VAR section	VAR_INPUT, VAR_OUTPUT, etc.
Has body?	Yes	No. It may define only methods
Can an instance of it be called?	Yes	No. Only the methods of a class may be called

The standard also defines classical function blocks and **introduces object-oriented FBs**.

3.1.2 Methods

Methods divide the **class** or **function block** into smaller functions that can be executed upon call. They will only work with the data they need, and they will ignore any redundant data that may exist in a certain function block.

Methods can access and manipulate the main class's internal variables, but they can also use variables of their own that cannot be accessed by the main class (unless they're output variables).

Also, methods are a much more efficient way of running a program because, by dividing a function into various methods, the user avoids running the whole POU every single time, running only small portions of code whenever there is a need for them to be called. This is a very good way of avoiding errors and data corruption.

Methods also have a name, which means that these portions of code can be identified by their purposes instead of the variables they manipulate, thus enhancing code reading and troubleshooting.

Figure 3.1 shows a simple example of a "Door" function block with two methods "openDoor" and "closeDoor". As it is possible to understand, the code is easy to read and understand: if there is a knock on the door, the door opens, otherwise, it closes.

```
IF knockOnDoor THEN
    openDoor();
    ELSE
    closeDoor();
END_IF
```

Figure 3.1 - "Door" FB calling its methods

Abstraction plays an important role here – if programmers wish to implement the code to open the door multiple times, they only need to call the method. Troubleshooting also becomes simpler – if the door is having problems while closing, then the programmer doesn't need to hunt for every instance of the code, they just need to check the "closeDoor" method.

Unlike the base class, methods use the controller's temporary memory – data is volatile, as their variables will only keep their values while the method is being executed. If values are supposed to be kept between execution cycles, then the variable should be stored in the base class or in some other place that will retain values from one cycle to the other (such as the global variable list).

Figure 3.2 shows a method incrementing a variable instantiated in its main class and an internal variable of its own every time it is run. Both variables are incremented 3 times, yet, their final result is different:

<table>
<tr><td rowspan="4">Function Block
Internal_FB_Var : INT := 0

Method
Internal_Meth_Var : INT := 0</td><td colspan="2">Method Implementation Code
1 Internal_FB_Var := Internal_FB_Var + 1;
2 Internal_Meth_Var := Internal_Meth_Var + 1;</td></tr>
<tr><td>After 1 Run</td><td>Internal_FB_Var = 1
Internal_Meth_Var = 1</td></tr>
<tr><td>After 2 Runs</td><td>Internal_FB_Var = 2
Internal_Meth_Var = 1</td></tr>
<tr><td>After 3 Runs</td><td>Internal_FB_Var = 3
Internal_Meth_Var = 1</td></tr>
</table>

Figure 3.2 - Memory usage by classes/FBs and methods

3.1.3 Properties

Properties are major variables of a class. They can be used as an alternative to regular class or function block I/O [14].

Properties have “Get” and “Set” methods that allow variables to be accessed and/or changed:

- Get - Method that returns the value of a variable;
- Set - Method that sets the value of a variable.

By deleting the “Get” or “Set” method, a programmer can make properties “write-only” or “read-only”, respectively.

Since these are methods, it means that properties can:

- Have their own internal variables;
- Perform operations before returning its value;
- The returned variable doesn’t need to be attached to a particular input or output (or internal variable) of the POU, it can return a value based on a certain combination of its variables;
- Be accessed upon event instead of being checked in every execution cycle.

3.1.4 Access Specifiers

Access specifiers give programmers more control over who can access which data.

Methods and properties can be **public**, **protected** or **private**. If the user doesn’t specify the access type, then the method will be public by default:

- Public methods can be accessed from within the class, from the inherited classes and from the main program.
- Private methods can only be accessed from within the class, they can’t be accessed from inherited classes or the main program.
- Protected classes can be accessed from within the class and from inherited classes, but they can’t be accessed from the main program.

Variables can also be **internal**, accessible only from the inside of a certain **namespace** (i.e., any POU, method or property).

Table 3.2 - Access Specifiers: who can access methods

Access Specifier	Namespace	Defining POU	Derived POU	Main Program
Public (Default)	A	A	A	A
Protected	A	A	A	NA
Private	A	A	NA	NA
Internal	A	NA	NA	NA

A – Accessible; NA – Not Accessible.

3.1.5 Inheritance

Inheritance is a mechanism that allows POUs to be **extended**. Extended or **inherited classes** acquire some or all the properties and methods of the parent class and implement additional properties and methods. Inherited function blocks may also inherit its parent’s body code implementation.

Figure 3.3 shows how inheritance works – classes and function blocks may extend classes, but only function blocks can extend other function blocks. Classes have no body code implementation, hence why they can't extend function blocks.

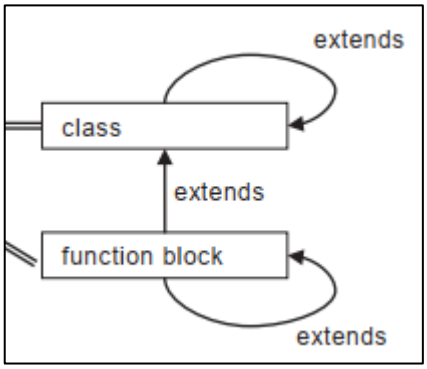


Figure 3.3 - How inheritance can be used to extend classes and function blocks [13]

Inheritance is very useful for the creation of classes which share the same basis of operation. Without it, POUs with small differences had to be created separately, demanding common code to be copied and pasted between POUs.

OOP allows programmers to create general base classes that can be applied to a large set of components with small differences. Inherited classes can then be created to account for those differences without influencing the main class.

Deep inheritance is possible: a class can be inherited from a class that is inherited from other class. However, deep inheritance with multiple levels of extension is not recommended, as it can generate lots of problems (i.e., the same class shouldn't be extended more than twice).

Multiple inheritance is **not** supported, the same class can have multiple extensions, but it can't be inherited from two parent classes. Figure 3.4 shows correct (left) and incorrect (right) uses of inheritance:

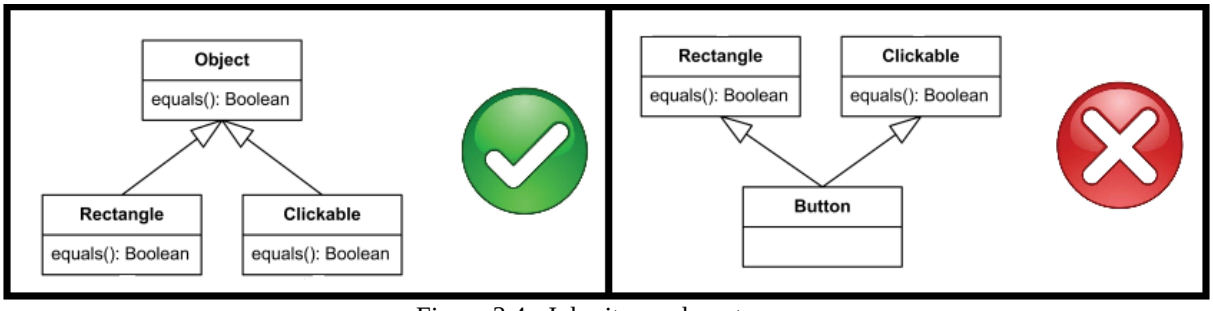


Figure 3.4 - Inheritance: how to use

Inherited POUs can **override** parent classes' methods and body code, which means that they may change the properties and methods they inherited without influencing the parent class.

Sometimes, the programmer may want to relate classes that do not have any type of relationship or dependency, which renders inheritance inadequate for the case – classes may have methods with the same name but with different implementations. In this case, the implementation of an **interface** - a class with methods and properties without implementation – is the solution.

3.1.6 Interfaces

An interface is a class that contains methods and properties without implementation. This interface can then be implemented in any class, but that class must implement all its methods and properties.

While inheritance is a “is a” relationship, interfaces can be described as a “behaves as” or a “has a” relationship [15].

Interfaces are objects that allow multiple different classes to have something in common with less dependencies. Classes and function blocks can **implement several different interfaces**. One can think of interface’s methods and properties as actions that mean different things depending on who is executing them. For instance, the word “Run” means “move at a speed faster than a walk” for a human being, but it means “execute” for computers.

Classes or function blocks that share no similarities may implement the same interface. In this case, the implementation of the methods in each class can be totally different. This opens up lots of powerful programming approaches:

- POUs can call an interface to execute a method or access a property, not knowing which class or FB it is dealing with or how it is going to execute the operation. The interface then points to a class or function block that implements the interface and the operation is executed;
- Programmers can create easily customizable switch cases using polymorphism.

3.1.7 Polymorphism

Polymorphism is used to change the object type at runtime. It can either be used with a base class and its inherited classes or with interfaces and the classes that implement them.

The way that polymorphism works with inheritance is very simple: there is a variable (**pointer** or **reference**) that points to a certain **base class**. During runtime, that pointer can alternate between classes if their type is either the base class or one of its inherited classes.

Working with interfaces, the variable type will be the **interface** itself. This is very powerful because the user can alternate between classes that are completely different as long as they implement the same interface. Interface’s methods perform different tasks depending on the class the interface is pointing to. Note that the **same variable cannot alternate between different interfaces**.

3.2 PLC Object-Oriented Programming Tools

As stated earlier in this document, special software development tools need to be used in order to implement the programming concepts introduced so far. The IEC 61131-3 Standard allows two basic concepts for the implementation of object-oriented programming:

- The use of OOP with object-oriented function blocks;
- OOP implementation based on classes.

The following section surveys two programming frameworks based on these approaches.

3.2.1 The CODESYS framework

CODESYS [16] is a development system for programming controller applications according to the international industrial standard IEC 61131-3. It is developed and marketed by the German software company 3S-Smart Software Solutions located in Kempten, Germany. The most recent version allows the user to program controllers using OOP concepts while providing the support that is needed to implement them.

CODESYS supports the use of function blocks, methods and interfaces defined by the IEC 61131-3 International Standard. However, CODESYS doesn't support the **class** concept and all the concepts related to it. It does support **object-oriented function blocks**. It offers extensive debugging functionalities such as variable monitoring and breakpoint setting.

Figure 3.5 shows the CODESYS' IEC 61131-3 Compliance Table [17]:

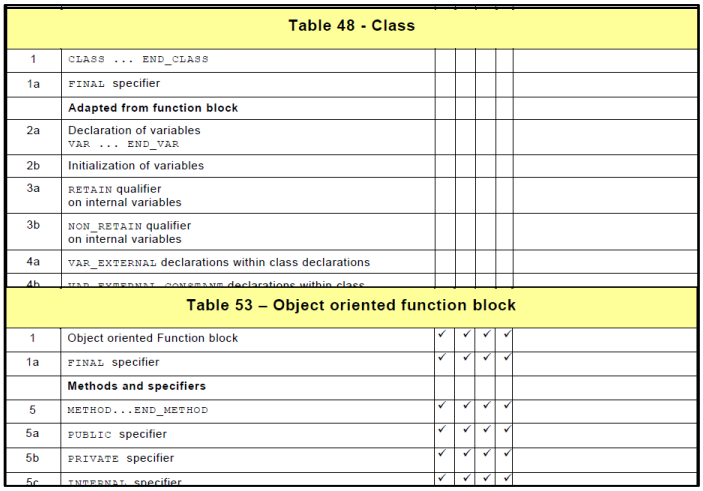


Table 48 - Class						
1	CLASS ... END_CLASS					
1a	FINAL specifier					
	Adapted from function block					
2a	Declaration of variables VAR ... END_VAR					
2b	Initialization of variables					
3a	RETAIN qualifier on internal variables					
3b	NON_RETAIN qualifier on internal variables					
4a	VAR_EXTERNAL declarations within class declarations					
4b	VAR_EXTERNAL CONSTANT declarations within class					
Table 53 – Object oriented function block						
1	Object oriented Function block	✓	✓	✓	✓	
1a	FINAL specifier	✓	✓	✓	✓	
	Methods and specifiers					
5	METHOD...END_METHOD	✓	✓	✓	✓	
5a	PUBLIC specifier	✓	✓	✓	✓	
5b	PRIVATE specifier	✓	✓	✓	✓	
5c	INTERNAL specifier	✓	✓	✓	✓	

Figure 3.5 - CODESYS compliance tables: doesn't support classes but supports object-oriented function blocks

Many PLC manufacturers (e.g., Schneider Electric or Beckhoff) use the CODESYS framework in their application design software.

3.2.2 Tools from Siemens

Siemens provides three software platforms that conform to IEC 61131-3:

- TIA Portal (Totally Integrated Automation Portal)/STEP 7 – doesn't support OOP;
- SIMOTION – supports object-oriented programming from V4.5 onwards.

TIA Portal's function blocks aren't object-oriented, but some of the functionalities that characterize OOP can still be emulated. However, its use of UDTs (user-defined types) is very limited when compared to environments that support object-oriented programming - e.g., arrays and structured data types cannot have UDTs as elements.

SIMOTION is a motion control system developed by Siemens. It has been marketed since 2002 and used in many kinds of machines. It is used for applications in which motion control plays a central role. Starting from version 4.5, SIMOTION fully supports all the concepts introduced by the IEC 61131-3 Standard regarding object-oriented programming.

However, Braun and Horn [18] state that SIMOTION chose not to implement the use of the object-oriented function blocks defined by the IEC. SIMOTION uses classes in order to implement OOP. Figure 3.6 shows an example of a “Counter” Class created in SIMOTION:

```
CLASS COUNTER
    VAR
        CV:INT; // Current value of counter
    END_VAR

    VAR OVERRIDE
        MAX_Val:INT := 100;
        MIN_Val:INT := 0;
    END_VAR

    METHOD PUBLIC UP: INT // Method for count up by inc
        VAR_INPUT
            INC:INT:=1;
        END_VAR
        VAR_OUTPUT
            QU:BOOL;
        END_VAR
        // Upper limit detection
        IF CV <= (MAX_Val - INC) THEN
              CV := CV + INC; // Count up of current value
            QU := FALSE;
        ELSE
              QU := TRUE;    // upper limit reached
        END_IF;
        UP := CV; // Result of method
    END_METHOD
END_CLASS
```

Figure 3.6 - Example of a "Counter" Class [18]

SIMOTION [18] also defines function blocks with methods without inheritance or overriding of methods. Figure 3.7 shows an example of a function block with a method.

```
FUNCTION_BLOCK FBValve43
    VAR_INPUT
        Mode         : BOOL;
        EndPos       : BOOL;
        StartPos     : BOOL;
        Forw         : BOOL;
        Backw        : BOOL;
    END_VAR

    VAR_OUTPUT
        QForw        : BOOL;
        QBackw       : BOOL;
    END_VAR

    VAR
        boMode           : BOOL;
        boEndPos         : BOOL;
        boStartPos       : BOOL;
        boForward        : BOOL;
        boBackward       : BOOL;
        boMoveForward    : BOOL;
        boMoveBackward   : BOOL;
    END_VAR

    METHOD mForw // Method move forward
        IF NOT boMode THEN // Jog mode
            IF boForward AND NOT boBackward THEN
                boMoveForward    := TRUE;
                boMoveBackward   := FALSE;
            ELSE
                boMoveForward    := FALSE;
            END_IF;
        ELSE    // Automatic mode
            IF (boForward OR boEndPos) AND NOT boBackward THEN
                boMoveForward    := TRUE;
                boMoveBackward   := FALSE;
            END_IF;
        END_IF;
    END_METHOD
END_FUNCTION_BLOCK
```

Figure 3.7 - Example of a “FBValve43” function block with methods [18]

3.3 Concluding Remarks

This chapter addressed the standards and resources that allow PLCs to be programmed using object-oriented programming approaches. The concepts that were introduced by the latest edition of the IEC 61131-3 international standard were briefly introduced to better understand the capabilities of this new programming approach.

In the next chapter, these concepts and technology will be put into practice in order to evaluate how useful they can become.

4 Practical Evaluation of Object-Oriented PLC Programming

The creation of applications to solve practical automation problems is very important to better understand the capabilities of this new programming approach. To validate the developed applications, simulated scenarios were created using Factory IO's [19] virtual environments.

Access to real systems is often very difficult. Simulated scenarios allow programmers to test their applications without having to entail the costs and risks associated with the creation of real systems.

CODESYS 3.5 SP10 Patch 1 was the software development environment used to create the control applications.

4.1 Training Environment

Factory IO is an increasingly popular 3D factory simulation software that allows for original user-defined or common industrial scenarios to be created in a simulation environment. It is compatible with any PLC, making a great tool for PLC training [19].

Figure 4.1 - Example of a Factory IO custom scenario

Factory IO provides a great set of industrial parts, including sensors, conveyors, elevators, stations and many others. Most parts include analog and digital I/O. So, it is the perfect environment to safely and inexpressibly develop and test PLC controlled industrial applications.

Factory IO has two modes of simulation:

- Edit – to create and edit the scenario;
- Run – to test the created scenario.

To hop between these modes, Factory IO provides the buttons shown in Figure 4.2, which can be found on the top right corner of the software's user interface:

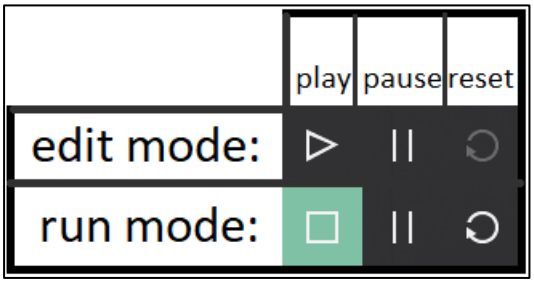


Figure 4.2 - Left: System is stopped; Right: System is running

The first button, the "**play**" button, starts the simulation. It is a toggle button which changes the its appearance depending on the simulation mode – in edit mode, the button looks like a "play" button and when the system is running, the button is subbed by a stop" button.

The second button, the "**pause**" button, pauses the running simulation. Pausing the simulation in run mode blocks all movement, even if users want to manually move components from their places, it won't be allowed. It is also a toggle button that can be activated in edit mode, so that the system can be started in pause mode.

The third button, the "**reset**" button, resets the simulation to its original state. Every item that may have been inserted is removed and every component that was moved in the simulation returns to its original position. It is a momentary button, as it isn't required to stay activated for long periods of time. It may only be activated during the simulation.

All buttons may be used as Inputs and Outputs of the PLC application.

Additional info on Factory IO can be found on their website [19].

4.2 Case Studies

This section features four simple case studies to explain how OOPs features can be applied to Factory IO's components:

- A generic conveyor, such as the one that was exposed in Figure 4.1;
- A conveyor that incorporates a data buffer capable of registering the ID of the items it carries;
- A conveyor scale, which is a specific conveyor that can measure the weight it is carrying;
- A sorting station, which is a generic conveyor that has an attached actuator that removes items from it.

4.2.1 A Generic Conveyor

This case study features a simple unidirectional belt conveyor that is meant to transfer parts from one side to the other. A function block called "Conveyor" was developed to control this

component. The function block divides its tasks between its body code implementation and its methods.

The developed scenario includes four components:

- **Belt Conveyor**: a unidirectional digital belt conveyor that can transport items (or parts). Figure 4.3 shows a conveyor that is able to transport parts from point A to point B. The arrow indicates the direction of the movement:

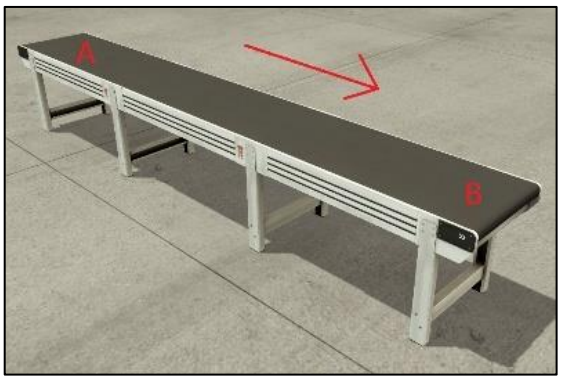


Figure 4.3 - Unidirectional digital belt conveyor

- **Emitter**: an emitter is a virtual component that simulates the insertion of items into a system. Figure 4.4 show an example of an Emitter:

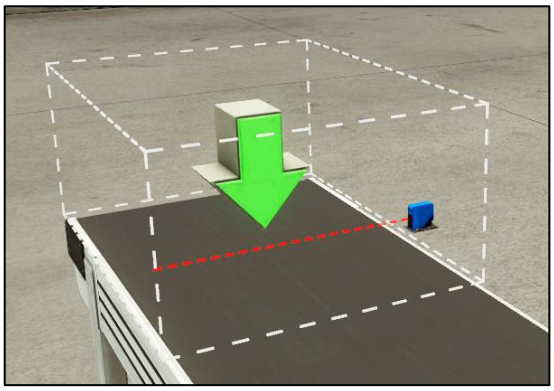


Figure 4.4 - Example of an Emitter: box with green arrow

- **Diffuse Sensors**: These diffuse photoelectric sensors can detect any solid object. Figure 4.5 shows a diffuse sensor detecting a box:

Figure 4.5 - Diffuse sensor detecting a box

- **Chute Conveyor**: Straight chute conveyor, mostly used to dispatch items from belt conveyors. Figure 4.6 shows a chute conveyor dispatching a large box:

Figure 4.6 - Chute Conveyor dispatching an item

4.2.1.1 Description of the First Scenario

The first scenario that was developed in this case study features a belt conveyor that transports items from the side where they are inserted to a chute conveyor that dispatches them. Figure 4.7 shows the assembly of this scenario in Factory IO:

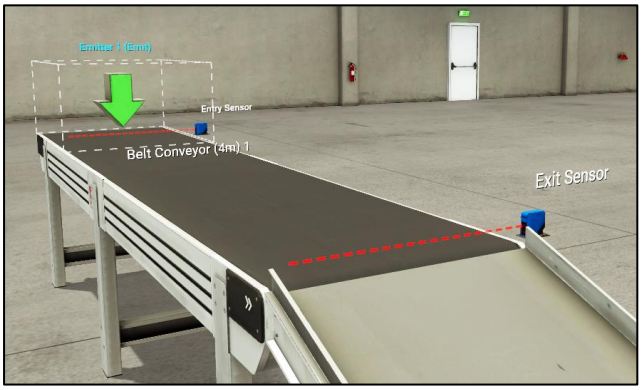


Figure 4.7 - Example of a simple unidirectional belt conveyor

- **Components**:
  - 1 Belt Conveyor;
  - 1 Emitter;
  - 2 Diffuse sensors;
  - 1 Chute Conveyor.
- **Scenario**:
  - The emitter inserts items on the conveyor;
  - The entry sensor detects the arrival of a new item;
  - The conveyor moves if it is carrying at least one item, otherwise it remains still;
  - As soon as an item leaves the emitter, the emitter inserts another one;
  - The exit sensor detects items' departures.
- **Aim**: Automatic control of a conveyor.
- **Initial State**: Conveyor with no items.
- **Manual Procedures**: Press the start button to turn the system on.
- **Used OOP features**: Methods.

4.2.1.2 Proposed Solution for the First Scenario

Table 4.1 describes the “Conveyor” function block’s variables and methods.

Table 4.1 - Description of the "Conveyor" function block

Conveyor			
Variables			
Variable	Name	Type	Description
Input	bEntrySensor	BOOL	Sensor at the beginning of the conveyor.
	bExitSensor	BOOL	Sensor at the end of the conveyor.
Output	bRoll	BOOL	Signals the conveyor to roll.
Internal	iMaxParts	INT	Capacity of the conveyor.
	Counter	CTUD	Counter the number of parts on the conveyor
Methods			
Run()	Allows the conveyor to work.		
Stop()	Keeps the conveyor from working.		
Reset()	Resets the conveyor’s default parameters.		

Figure 4.8 shows a GRAFCET [20] that explains how the conveyor works in this specific scenario.

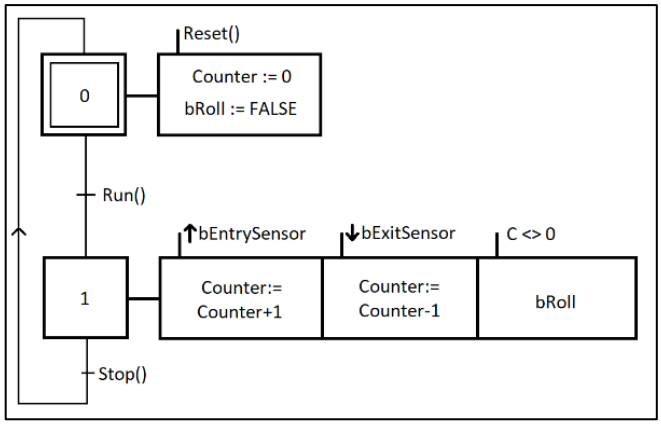


Figure 4.8 - Conveyor’s functional GRAFCET

By analyzing the GRAFCET shown in Figure 4.8, it is possible to identify two types of actions:

- Event triggered actions – e.g., the counter is incremented when “bEntrySensor” exhibits a rising trigger;
- Continuous actions – e.g., turning “bRoll” on.

It is important to understand which actions should be assigned to the body and methods of the function block.

As stated in section 3.1.2, methods use temporary memory, whereas function blocks use static memory. This means that methods are more suitable for performing event triggered actions whereas the body of the FB is more suitable for performing continuous actions. However, this isn’t always true.

Table 4.2 describes where code should be implemented:

Table 4.2 - Where code should be implemented

Conveyor		
Type of Action	Description	Namespace
Continuous	**Setting** the "bRoll" Output.	Body of the FB
Event Triggered	**Incrementing/Decrementing** the counter.	
	Starting/Stopping the system allows the conveyor to work or keeps it from working.	Methods
	Resetting the system's default parameters such as resetting the counter's current value to 0.	

Counters increments and decrements are event triggered actions that are directly included in the counter function block that CODESYS provides. For that reason, adding a method to increment and another method to decrement the counter only adds unwanted complexity to the system.

4.2.1.3 Description of the Second Scenario

Before moving on to the next case study, a new functionality is going to be added to this general conveyor: the ability of transferring items to other components and checking if they're full. Figure 4.9 shows a scenario that features two conveyors: parts are inserted onto the first conveyor, which transfers them to a second conveyor that is responsible for carrying them to the chute conveyor.

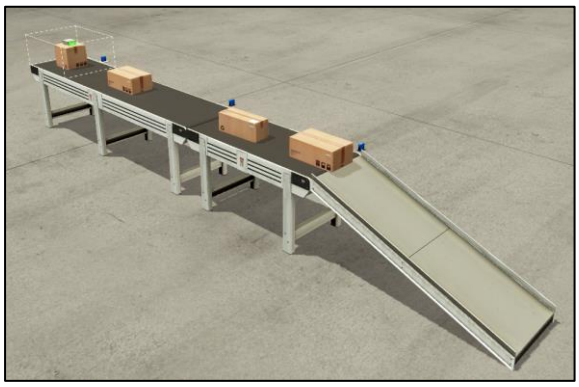


Figure 4.9 - Second scenario of the case study: transferring parts between conveyors

- **Components**:
  - 2 Belt Conveyors;
  - 1 Emitter;
  - 3 Diffuse sensors;
  - 1 Chute Conveyor.
- **Scenario**:
  - As items from the first conveyor reach its end, the second conveyor detects the arrival of a new item;
  - If the second conveyor is already carrying two items, it signals that it is full;
  - If the second conveyor is full and one item reaches the end of the first conveyor, then it will stop and wait for the next conveyor to free up space.
- **Aim**: Automatic control of a line of conveyors.
- **Initial State**: Conveyors with no items.
- **Manual Procedures**: Press the start button to turn the system on.
- **Used OOP features**: None.

4.2.1.4 Proposed Solution for the Second Scenario

Table 4.3 describes the “Conveyor” function block’s new variables.

Table 4.3 - Description of the "Conveyor" function block's additional variables

Conveyor			
Variables			
Variable	Name	Type	Description
Input	bNextCompFull	BOOL	Signals if the next component is full
Output	bFull	BOOL	Signals if the conveyor is full.

Figure 4.10 shows the changes and additions that were made to the functional GRAFCET shown in Figure 4.8:

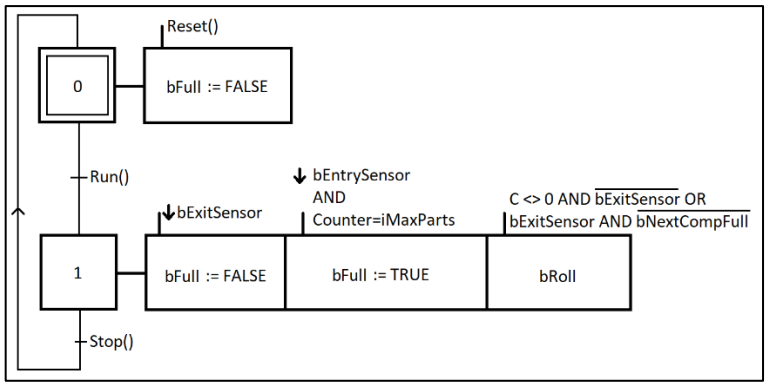


Figure 4.10 - "Simple Conveyor" functional GRAFCET

In this case, the management of the “bFull” variable is done in the body of the function block despite being set/reset by events. Since its conditions are clearly defined, the management of this variable can easily be done with a latching relay.

The next case study will use this conveyor and **extend** its functionalities by adding a data buffer that records the items it carries.

4.2.2 Conveyor with FIFO

This case study features a conveyor that incorporates a data buffer which records every part it carries at a certain moment. The "Conveyor" function block was **extended** in order to **encapsulate** a "FIFO" function block.

Before explaining how the scenario works, the identification system and the "FIFO_INT" function blocks are going to be introduced. Figure 4.11 shows a box identification system attached to an emitter and a conveyor:

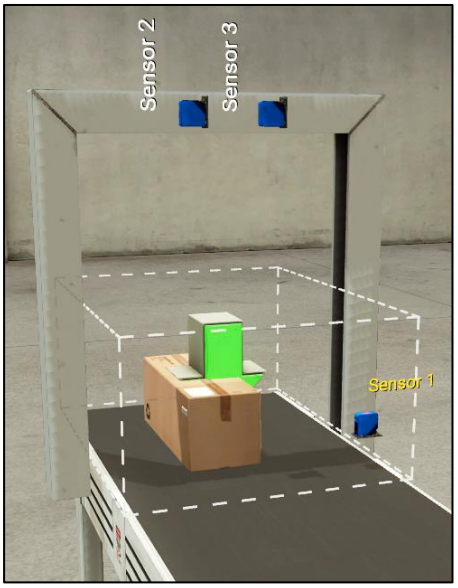


Figure 4.11 - Box Identification System attached to an emitter and a conveyor

The Box Identification System features 3 sensors. Table 4.4 shows the different combinations of their signals that identify different boxes:

Table 4.4 - How the Box Identification System works

Box Identification System			
Box	Sensor 1	Sensor 2	Sensor 3
Small	TRUE	FALSE	FALSE
Medium	TRUE	FALSE	TRUE
Large	TRUE	TRUE	TRUE

The conveyors encapsulate a "FIFO INT" function block. FIFO (First In, First Out) organizes and manipulates a data buffer, where the oldest entry (or "head") of the queue is processed first. Table 4.5 describes the "FIFO INT" function block that was used in the case study:

Table 4.5 - Description of the "FIFO INT" function block

FIFO INT					
Methods	Description	I/O	Type	Type	Description
Write()	Writes data on the buffer.	Input	DataIn	INT	Data to be inserted in the buffer.
Read()	Reads data from the buffer	Output	DataOut	INT	Data to be removed from the buffer.
Reset()	Resets the buffer's default settings.				

4.2.2.1 Description of the Scenario

Figure 4.12 shows the second scenario of the previous section with some additional components. An identification system identifies boxes by their size at the beginning of the first conveyor. As a box is being dispatched, a light is turned on depending on their size. In this case, a medium box is being dispatched:

Figure 4.12 - Medium box being dispatched

- **Components**:
  - 2 Belt Conveyors;
  - 1 Emitter;
  - 5 Diffuse sensors;
  - 1 Chute Conveyor;
  - 1 Bracket;
  - 1 Stack Light with 3 Light indicators (Red, Yellow and Green).
- **Scenario**: The identification system identifies boxes by their size as they enter the first conveyor. The conveyors store the IDs of the parts they carry. The first conveyor transfers a part to the second conveyor along with its ID. As the box is being dispatched to the chute conveyor, the light that corresponds to its size is turned on:
  - Small: Green;
  - Medium: Yellow;
  - Large: Red.
- **Aim**: Incorporation of a monitoring system in a Conveyor.
- **Initial State**: Conveyor with no items.
- **Manual Procedures**: Press the start button to turn the system on.
- **Used OOP features**: Inheritance, Encapsulation, Methods, Delegation.

4.2.2.2 Proposed Solution for the Scenario

Table 4.6 describes the additional variables and methods that the "Conveyor with FIFO" FB implements when related to the "Conveyor" FB described in the previous case study:

Table 4.6 - Description of the "Conveyor with FIFO" function block

Conveyor with FIFO			
Variables			
Variable	Name	Type	Description
Input	iPartTypeIn	INT	ID of the part that is entering the conveyor.
Output	iPartTypeOut	INT	ID of the part that is exiting the conveyor.
Internal	Buffer	FIFO_INT	Array where the parts that the system is carrying are recorded.
Methods			
Reset()	Does exactly what the "Conveyor" FB's Reset() method does, but it also resets the buffer.		

Figure 4.13 introduces the additional features that the "Conveyor with FIFO" has in relation to the previous case study:

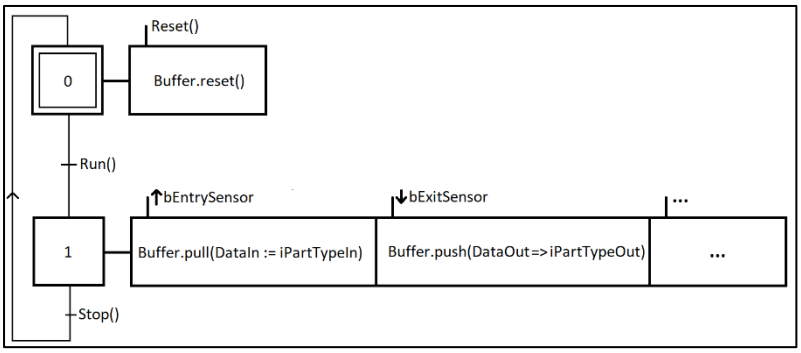


Figure 4.13 - "Conveyor with FIFO" functional GRAFCET

The following OOP features were used:

- **Inheritance** – extending the "Conveyor" FB to create the "Conveyor with FIFO" FB. Figure 4.14 shows a UML representation of the extension:

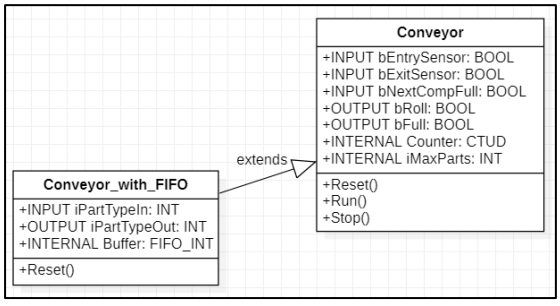


Figure 4.14 - UML representation of the "Conveyor with FIFO" extension

- **Encapsulation** – a user-defined function block, "FIFO_INT" was encapsulated in the extension;
- **Methods** – the "FIFO_INT" function block has methods of its own, which are responsible for managing the array;

- **Delegation** – A function block can use the (public) methods of the blocks that it incorporates. FIFO_INT's methods – Write(), Read() and Reset() – are called **delegates**. From the outside, it might seem like the Conveyor is storing IDs in the array, but it is, in fact, the "FIFO_INT" FB that is performing that task. Delegation is what differentiates encapsulation in the latest edition of the IEC 61131-3 standard, since previous editions didn't support the implementation of methods.

Using Inheritance to extend the "Conveyor" FB and encapsulating the "FIFO_INT" FB is a much simpler, easier and quicker way to merge both POUs than using only encapsulation:

- The new function block would require the definition of all the I/O of the "Conveyor" and the "FIFO_INT" FBs, which would require a lot of copying and pasting;
- It is possible to say that the "Conveyor with FIFO" **is a** "Conveyor". In this case, inheritance can be applied to extend the "Conveyor" function block, requiring only the additional features, therefore reducing copying and pasting;
- Take into account that a conveyor with a buffer **is not** a buffer. Extending the "FIFO" FB and adding the "Conveyor" FB would make no sense.
- Pure encapsulation should be limited to cases where the new function block doesn't classify as any of its constituents (e.g., a machine **is not a** motor or a temperature sensor).

Figure 4.15 shows how clean and simple the variable instantiation of the new function block becomes, as it doesn't require the redefinition of the Conveyor's I/O. However, when the "Conveyor with FIFO" FB is called on by another POU, it may access all the I/O of the "Conveyor" FB:

```
1  FUNCTION_BLOCK Conveyor_with_FIFO EXTENDS Conveyor
2  VAR_INPUT
3      partTypeIn : INT;
4  END_VAR
5  VAR_OUTPUT
6      partTypeOut : INT;
7  END_VAR
8  VAR
9      Buffer : FIFO_INT;
10 END_VAR
```

Figure 4.15 - "Conveyor with FIFO" function block instantiation

Figure 4.16 shows how simple it is to implement the additional code and how easy it is to read. In line 3, it is also possible to see the definition of a rising edge. It is, in fact, the extension overriding its parent since that rising edge has a different definition in the parent function block.

```
1  SUPER^();
2
3  r_entrySensor(CLK := entrySensor);
4
5  IF r_entrySensor.Q THEN
6      Buffer.write(DataIn := partTypeIn);
7  END_IF
8
9  IF f_exitSensor.Q THEN
10     buffer.read();
11 END_IF
12
13 Buffer(DataOut => partTypeOut);
```

Figure 4.16 - "Conveyor with FIFO" function block body code implementation

Inheritance can be used to extend a function block that defines a default set of behaviors for a set of components. Factory IO provides other types of unidirectional digital conveyors that can

perform additional tasks. The next section will show how OOP allows programmers to deal with small differences between similar systems.

4.2.3 Conveyor Scale

This case study features a Conveyor Scale that is able to measure the weight of the item it is carrying. The "Conveyor with FIFO" function block was extended in order to deal with the system's differences.

4.2.3.1 Description of the Scenario

Figure 4.17 shows a conveyor scale which features 3 diffuse sensors:

Figure 4.17 - Example of a Conveyor Scale

- **Components**:
  - 1 Conveyor Scale;
  - 3 Diffuse sensors.
- **Scenario**:
  - The conveyor scale receives an item from another conveyor;
  - As soon the 2nd sensor (the sensor in the middle of the Conveyor Scale) detects a part, the conveyor stops moving, since the measurement is inaccurate while the conveyor is moving;
  - After stopping for 1 second, the POU acquires the item's weight and resumes the movement.
- **Aim**: Measurement of an item's weight.
- **Initial State**: Conveyor with no items.
- **Manual Procedures**: Press the start button to turn the system on.
- **Used OOP features**: Inheritance, Methods.

4.2.3.2 Proposed Solution for the Scenario

Table 4.7 describes the additional variables and methods that the "Conveyor with FIFO" FB implements when related to the "Conveyor" FB described in the previous case study:

Table 4.7 - Description of the "Conveyor Scale" function block

Conveyor Scale			
Variables			
Variable	Name	Type	Description
Input	mSensor	BOOL	Sensor in the middle of the conveyor.
	iWeightLimitInKg	INT	Conveyor scale's weight limit (Kg).
	iVoltLimit	INT	Conveyor scale limit in Volts.
	wWeightInVolts	WORD	Analog signal acquired from the scale.
Output	wWeightInKg	WORD	Scaled/Converted weight value.
Internal	onDelay	TON	On Delay timer
Methods			
Reset()	Does exactly what the "Conveyor with FIFO" FB's Reset() method does, but it also resets additional variables.		
Pause()	Pauses the conveyor in order to wait for an accurate measurement.		
Convert()	Scales/converts the weight value from volts to kilograms.		

The conveyor scale works just like a regular Conveyor, or a Conveyor with FIFO in this case, but it incorporates additional I/O and code to cope with the additional physical features. Figure 4.18 introduces the additional features that the "Conveyor Scale" FB has in relation to the "Conveyor with FIFO" FB:

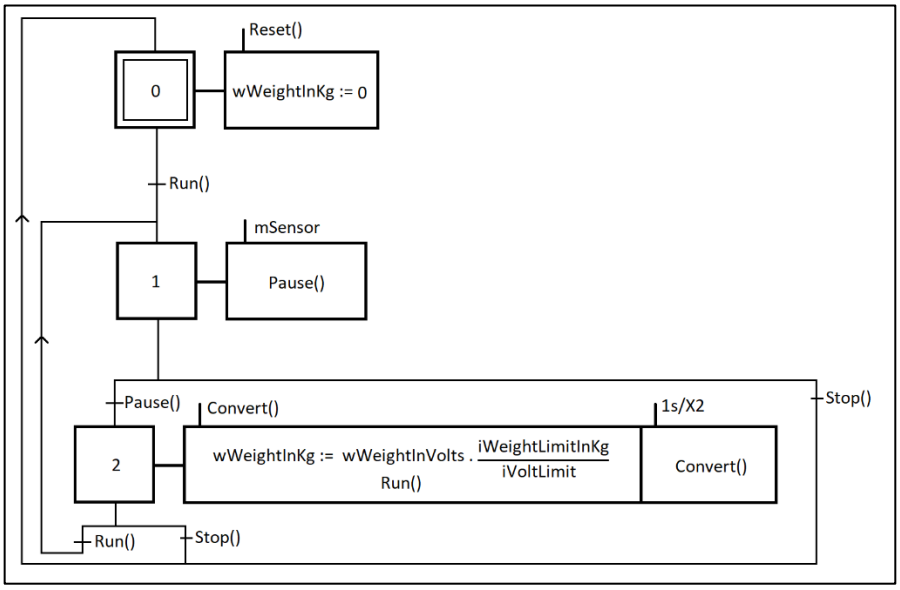


Figure 4.18 - "Conveyor Scale" functional GRAFCET

The following OOP features were used:

- **Inheritance** – extending the "Conveyor with FIFO" FB in order to create the "Conveyor Scale" FB. Figure 4.19 shows a UML representation of the extension:

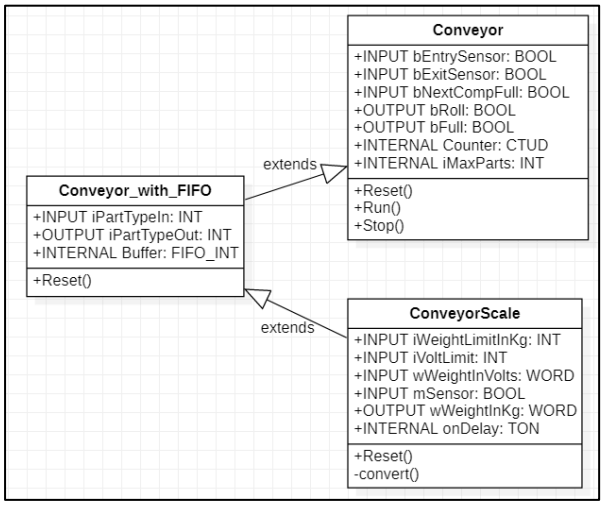


Figure 4.19 - UML representation of the "Conveyor Scale" extension

- **Methods** – the "Conveyor Scale" FB uses a new private method called "Convert()" to scale the value from volts to kilograms. The term "convert" was used instead of "scale" to avoid confusion. It also uses a method called "Pause()" in order to pause the movement for one second to get an accurate weight measurement.

Having a framework that defines the core properties and behaviors of a system can prove to be very important. As stated earlier, Factory IO provides other types of digital conveyors that have other functionalities, but the only thing that all conveyors have in common is the fact that **they are** conveyors.

Not having to reimplement the code over and over, or not having to encapsulate the "Conveyor" FB which requires having to redefine the I/O can prove to be a great advantage in the long run. Even if components have differences in relation to the parent function block, they can always override it, and if changes are applicable to every component, then they only need to be applied in the parent class.

The only problem that comes with inheritance is related with deep inheritance, which can bring up problems due to excessive overriding.

Inheritance is, thus, a very powerful resource when dealing with similar components. However, it is not the best answer when it comes to dealing with different components which may apparently have nothing in common. The next section addresses how programmers can link components with no dependencies.

4.2.4 Sorting Station: Conveyor with an Item Removing Actuator

This case study features two Sorting Stations: two conveyors with an actuator (a pusher or a pivot arm sorter) that removes parts from it. The "Conveyor with FIFO" FB was extended in order to create the "Sorting Station" FB which encapsulates the "itfItemRemover" Interface.

Using an interface, the sorting station can point to different actuators and order them to do the same task, even if they have no similarities. Components are only required to implement that

interface, which is an empty class. The implementation of the properties and methods they acquire can be completely different.

Before explaining the case study, the Interface and the actuators that implement it are going to be introduced. Table 4.8 describes the “itfItemRemoverInterface”:

Table 4.8 - Description of the “itfItemRemover” Interface

itfItemRemover		
Property	Type	Description
p_bBusy	BOOL	Signals if the actuator is eliminating an item.
Methods	Description	
RemoveItem()	Removes Item from the sorting station.	

The pneumatic Pusher sorter is equipped with a rod to push items and two reed sensors indicating the front and back limits. Table 4.9 describes the “Pusher” function block:

Table 4.9 - Description of the “Pusher” function block

Pusher			
Variables			
Variable	Name	Type	Description
Input	bBackLimit	BOOL	Signals if the Pusher is at its back limit.
	bFrontLimit	BOOL	Signals if the Pusher is at its front limit.
Output	bPush	BOOL	Pushes the rod.
In/Out	Conv	Conveyor	Points to the sorting station it is attached to.
Property	p_bBusy	BOOL	Signals if the actuator is eliminating an item.
Methods			
RemoveItem()	Removes Item from the sorting station.		

A Pivot Arm Sorter is a 45º power face arm diverter, powered by a gearmotor, equipped with a belt that helps to deviate the conveyed items onto the next part. The arm can rotate left or right according to the selected configuration. In this case, the arm only turns right. Table 4.10 describes the Pivot Arm Sorter function block:

Table 4.10 - Description of the “Pivot Arm Sorter” function block

Pivot Arm Sorter			
Variables			
Variable	Name	Type	Description
Output	bTurn	BOOL	Turns the Arm.
	bRoll	BOOL	Rolls the belt.
Internal	onDelay	TON	Points to the sorting station it is attached to.
Property	p_bBusy	BOOL	Signals if the actuator is eliminating an item.
Methods			
RemoveItem()	Removes Item from the sorting station.		

Figure 4.20 shows the “Pusher” FB and the “Pivot Arm Sorter” FB implementing the “itfItemRemover” interface:

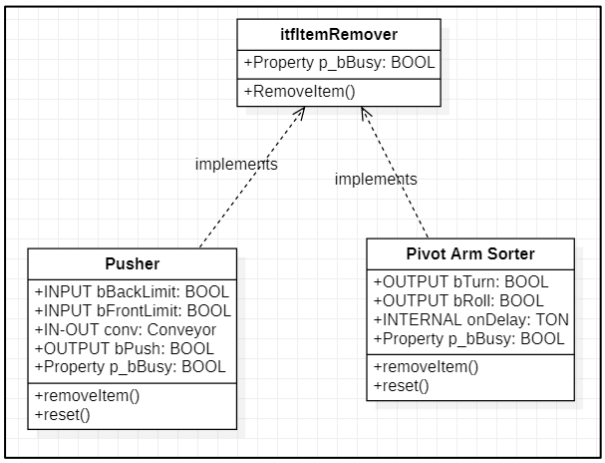


Figure 4.20 - UML representation of function blocks implementing the “itfItemRemover” interface

A Pusher and a Pivot Arm Sorter are two components that share no similarities but perform the same task. Yet, because of the implementation of the interface, the Sorting Station is able to order them to remove items as if they were the same actuator.

4.2.4.1 Description of the Scenario

Figure 4.21 shows two sorting stations: one with a Pusher (left) and one with a Pivot Arm Sorter (right):

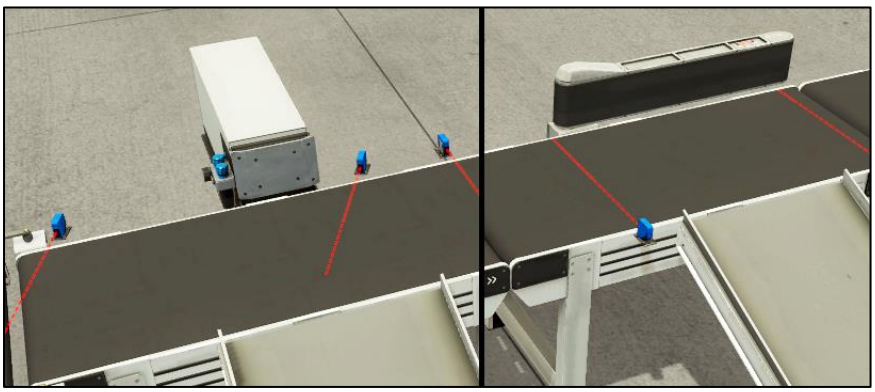


Figure 4.21 - Sorting Station: a conveyor with an actuator that removes parts from it (Pusher: left; Pivot Arm Sorter: right)

- **Components**:
  - 2 Belt Conveyors;
  - 1 Pusher;
  - 1 Pivot Arm Sorter;
  - 6 Diffuse sensors (3 in each Conveyor);
  - 2 Chute Conveyors.
- **Scenario**:
  - The sorting station receives an item from another conveyor;
  - As soon as it reaches the sensor of the actuator, it reads the part’s ID and if it matches any of the IDs that the operator wishes to eliminate, then it orders the actuator to eliminate the part.

- **Aim**: Incorporation of a communication framework between different components.
- **Initial State**: Conveyor with no items.
- **Manual Procedures**: Press the start button to turn the system on.
- **Used OOP features**: Inheritance, Encapsulation, Interfaces, Methods, Delegation.

4.2.4.2 Proposed Solution for the Scenario

Table 4.11 describes the additional variables and methods that the "Sorting Station" FB implements when related to the "Conveyor with FIFO" FB described in the previous case study:

Table 4.11 - Description of the "Sorting Station" function block

Sorting Station			
Variables			
Variable	Name	Type	Description
Input	aPartsToEliminate	ARRAY OF INT	Array that provides the ID of the parts to be eliminate.
	bActuatorSensor	BOOL	Detects parts passing by the actuator.
In/Out	Actuator	itfItemRemover	Interface that communicates with the actuator.
Methods			
Reset()	Does exactly what the "Conveyor with FIFO" FB's Reset() method does, but it also resets additional parameters.		

Figure 4.22 introduces the additional features that the "Sorting Station" FB has in relation to the "Conveyor with FIFO" FB:

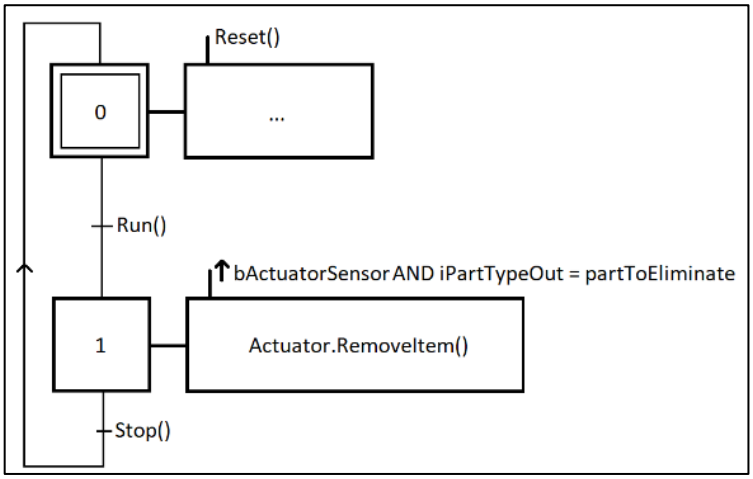


Figure 4.22 - "Sorting Station" functional GRAFCET

The following OOP features were used:

- **Inheritance –** extending the “Conveyor with FIFO” FB in order to create the “Sorting Station” FB. Figure 4.23 shows a UML representation of the extension:

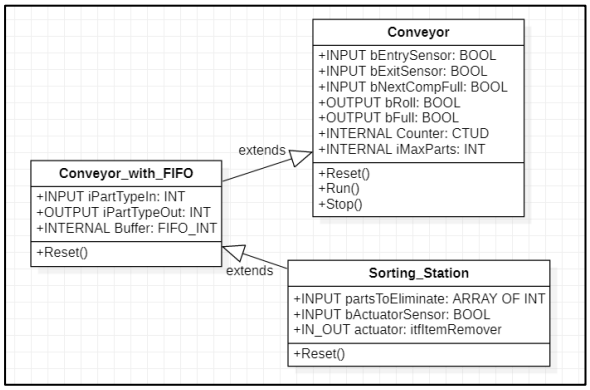


Figure 4.23 - UML representation of the "Sorting Station" extension

- **Encapsulation** – used to encapsulate the “itfItemRemover” interface in the new function block;
- **Interfaces** – the “itfItemRemover” interface is used by the FB to communicate with the function blocks that implement it (in this case, the Pusher and the Pivot Arm Sorter Function blocks);
- **Methods –** the interface forces the definition of a method called “RemoveItem()” on the function blocks that implement it;
- **Delegation** – the function block delegates the item removal task to the interface’s “RemoveItem()” method. The interface then proceeds to pass that order to the function block it is pointing to.

This standardized communication framework makes writing the code implementation easier, as the user only needs to attach the actuator to the sorting station. This is very powerful, as this:

- Makes linking function blocks much easier;
- Facilitates changing the actuator that is being used in a sorting station;
- Other types of actuators can be introduced as long as they implement the interface that was implemented by previous systems.

These advantages mean that if there are changes in the physical system that may render the actuators unsuitable for the task, then the programmer just needs to:

- Make sure that the new component implements the “itfItemRemover” interface;
- Instantiate it in the main program;
- Directly attach it to the Input of the “Sorting Station” FB.

This brings lots of advantages regarding future changes and data accessing, as well as it keeps the code clean and easy to read, which makes troubleshooting easier.

4.3 Personal overall analysis

Object-oriented programming is very adequate to create applications for PLC controlled industrial environments as it brings a lot of advantages regarding modularity, simplicity, ease of implementation, application of changes, data management, communication between components, along with various other advantages.

OOP provides extremely powerful features, but most importantly, it also aims to enhance the application's simplicity and modularity. The powerful features that OOP provides may not be applicable to every component or system, as it could just bring unwanted complexity. Programmers must aim for the lowest possible amount of complexity and dependencies when designing their applications.

Well-written object-oriented PLC applications can be very simple, easy to understand and powerful, but it requires lots of studying, knowledge and experience.

Programmers must fully master object-oriented programming before developing applications for real systems. There isn't much information about object-oriented PLC programming available online for free, however, there is a lot of information on other object-oriented programming languages such as C++, Java or Python, which proved to be a very helpful resource in order to grasp the concepts that were difficult to understand.

Poorly-written PLC applications can get confusing really quickly as the amount of dependencies increases or as code segregation and/or overriding is overdone. Troubleshooting can become a real "treasure hunt", as debuggers may need to search between different function blocks for the actual code implementation besides having to find the bug.

All in all, object-oriented programming isn't very easy to understand, especially for people with little to no programming knowledge. However, once one fully masters its concepts, OOP becomes a very programming approach tool to develop powerful and simple applications to control intelligent and versatile systems.

4.4 Concluding Remarks

This chapter addressed multiple small case studies that featured the creation of function blocks using object-oriented programming. It showed that the impact of OOP's features can be felt even in the simplest case studies.

The next section will address the improvement of a larger case study. A large industrial scenario will be analyzed and improved using object-oriented programming.

5 Improving Industrial Scenarios Using the Third Edition of the IEC 61131-3

Object-Oriented Programming as described in the IEC 61131-3 standard has brought a lot of changes to the PLC industry. This chapter aims to analyze if these changes bring clear advantages in relation to previous editions of the standard.

5.1 Previous work on IEC 61131-3 PLC programming

In order to analyze the impact of the features that were brought by the latest edition of the IEC 61131-3 standard, the "*Controlo Modular e Confiável de Sistemas Flexíveis de Automação*" (Modular and Reliable Control of Flexible Automation Systems) [21] dissertation was read.

Its core aim was to create standard solutions for general and complex automated production systems, as well as the creation of a function block library for ease of implementation in various simulated industrial scenarios, ensuring its correct behavior.

Custom function blocks were created for some parts provided by Factory IO V1.0, for direct implementation in case studies.

Applications were designed using CODESYS V3.5 SP5 Patch 2 and simulated scenarios were created using Factory IO. When the previous work was developed, CODESYS did not support object-oriented programming.

5.2 Previously Designed Scenario

The previous work featured the creation of non-object-oriented standard function blocks applied to some simulated industrial scenarios.

The choice of the scenario was based on the applicability of object-oriented programming on the control application in order for the changes to be clearly seen.

Before exposing the scenario that was studied, the components and their controlling function blocks are going to be explained in detail.

5.2.1 Introducing the Components and the Previously Designed POUs of the System

The previous work uses 5 POUs to control 5 types of components: Emitters, Removers, Roller Conveyors, Turntables and Chain Transfer Tables. This section will introduce these components and their POUs as they were described in the previous work. All images and tables in this section were taken from the previous work.

5.2.1.1 Emitter and Remover

The **Emitter** and **Remover** are POUs with just one output to order them to emit or remove, respectively. Figure 5.1 and Figure 5.2 show the emitter and remover available in Factory IO's previous edition. Table 5.1 and Table 5.2 describe both function blocks:

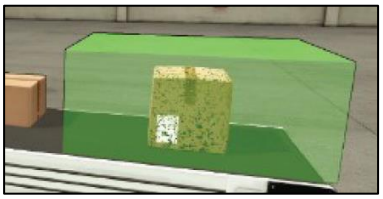


Figure 5.1 - Emitter available in the previous Factory IO edition

Table 5.1 - Emitter's Inputs and Outputs

Emitter			
Inputs		Outputs	
Name	Type	Name	Type
		Emit	BOOL

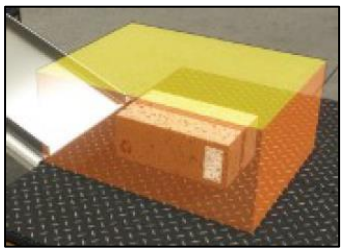


Figure 5.2 - Remover available in the previous Factory IO edition

Table 5.2 - Remover's Inputs and Outputs

Remover			
Inputs		Outputs	
Name	Type	Name	Type
		Remove	BOOL

5.2.1.2 Belt Conveyor Analog

The **Belt Conveyor Analog** is a POU that controls a conveyor's speed. Despite being called "Belt" Analog Conveyor, it can also control Roller Conveyors. This POU works like the "General Conveyor" FB designed in section 4.2.1, but it features an additional input variable, a WORD that sets the speed of the conveyor.

Figure 5.3 shows the type of Conveyor controlled in this scenario:

Figure 5.3 - Roller Conveyor available in the previous Factory IO edition

5.2.1.3 Chain Transfer Table and Turntable

The **Chain Transfer Table** and **Turntable** are the two pallet transferring mechanisms used in the scenario. Since these components haven't been introduced yet, the solution that was exposed in the previous work will be explained in depth.

Table 5.3 describes how a Chain Transfer Table works and introduces its inputs and outputs:

Table 5.3 - Chain Transfer Table: Description and I/O

Chain Transfer Table			
Description	The Chain Transfer Table is a mechanism that can transfer parts between a maximum of 4 directions. The mechanism has a set of rollers and chain belts that perform the transfer of an item – the rollers can send the item to the front or to the back and the chain belt can transfer them to the left or to the right. Figure 5.4 shows a Chain Transfer Table connected to 3 roller conveyors and its outputs (which share the name of the direction of the movement).		
Inputs		Outputs	
Name	Type	Name	Type
Sensor_north	BOOL	***Roll_pos***	BOOL
Sensor_south	BOOL	***Roll_neg***	BOOL
Sensor_east	BOOL	***Right***	BOOL
Sensor_west	BOOL	***Left***	BOOL

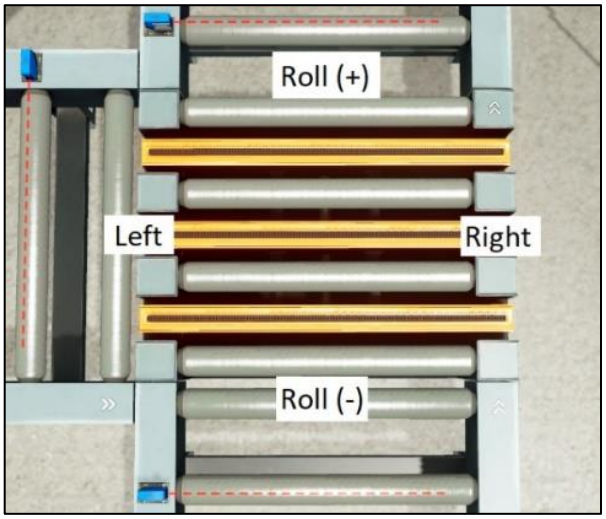


Figure 5.4 - Chain Transfer Table's Outputs (and directions)

The previous work stated that the Chain Transfer Table required two additional variables in order to be controlled. Table 5.4 defines those variables:

Table 5.4 - Chain Transfer Table's additional control variables

Chain Transfer Table			
Additional control variables			
Name	Type	I/O	Description
Loaded	BOOL	Input	Signals if the pallet is loaded.
Unloaded	BOOL		Signals if the pallet was unloaded.

Figure 5.5 shows the Chain Transfer Table's behavioral GRAFCET:

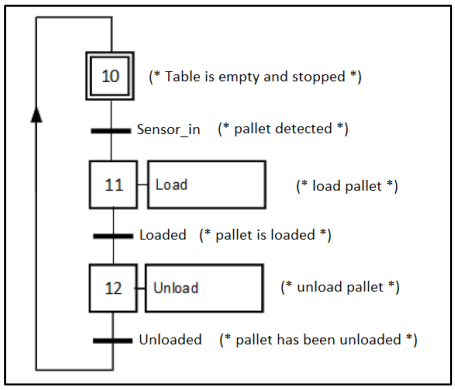


Figure 5.5 - Chain Transfer Table's behavioral GRAFCET

Table 5.5 describes how a Turntable works and introduces its inputs and outputs:

Table 5.5 - Turntable: Description and I/O

Turntable			
Description	The turntable is a mechanism that can transfer items between a maximum of four directions. It has a set of rollers on a tray that can turn 90 degrees. The rollers are bidirectional, which allows the system to transport parts from/to any of the four directions. This mechanism features 4 embedded sensors – 2 that detect the item's position and 2 that detect the position of the turntable. Figure 5.6 shows the position of the new I/O.		
Inputs		Outputs	
Name	Type	Name	Type
Sensor_north	BOOL	***Roll_pos***	BOOL
Sensor_south	BOOL	***Roll_neg***	BOOL
Sensor_east	BOOL	***Turn***	BOOL
Sensor_west	BOOL		
Sensor_0º	BOOL		
Sensor_90º	BOOL		
Front_limit	BOOL		
Back_limit	BOOL		

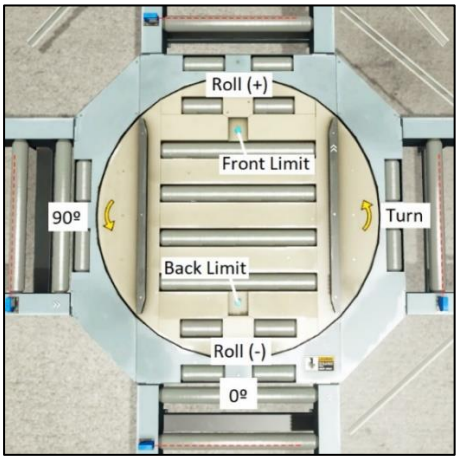


Figure 5.6 - Turntable's Outputs

Figure 5.7 shows the Turntable's behavioral GRAFCET:

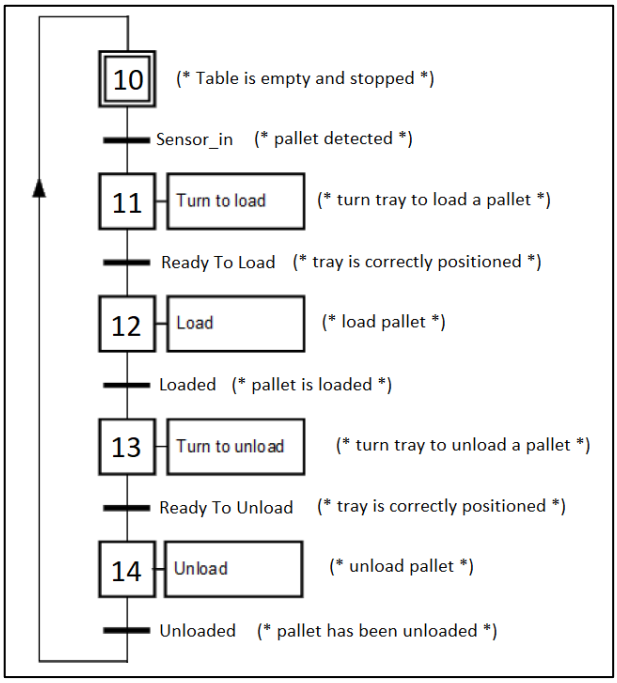


Figure 5.7 - Turntable's behavioral GRAFCET

Both tables receive a pallet that is then sent to the desired destination. As they can only transfer one part at a time, they must send busy signals to the adjacent components. They must also know if the components they're attached to are available to receive a part, so that an item isn't pushed onto a full conveyor. Table 5.6 describes the tables' common variables:

Table 5.6 - Tables' common variables

Tables			
Description	Both tables share some common variables, such as the destination of a part and busy signals (inputs and outputs).		
Control variables			
Name	Type	I/O	Description
Busy_in_north	BOOL	Input	Component's availability in each direction: TRUE – Component is busy; FALSE – Component is available.
Busy_in_south	BOOL		
Busy_in_east	BOOL		
Busy_in_west	BOOL		
Destination	INT		Pallet's destination: 1: Back; 2: Left; 3: Front; 4: Right.
Busy_in_north	BOOL	Output	Table's availability in each direction: TRUE – Table is busy; FALSE – Table is available.
Busy_in_south	BOOL		
Busy_in_east	BOOL		
Busy_in_west	BOOL		

5.2.1.4 Timeout Monitor

Even though it wasn't featured (at least directly) in the scenario, the author of the previous work created a function block to detect malfunctions in sensors and actuators.

A sensor may be malfunctioning when it doesn't detect an item that's in front of it or when it detects an item when, in reality, there's nothing close to it. Something similar can happen with an actuator, that is, remain in action when it is no longer required or not acting when required. A possible way of detecting this type of problems is through the implementation of a timeout monitoring function block: they emit a warning signal when a predetermined time (PT) limit for actuation of a sensor or actuator is exhausted.

Figure 5.8 shows temporal diagram that exemplifies the detection of a malfunction on an Actuator:

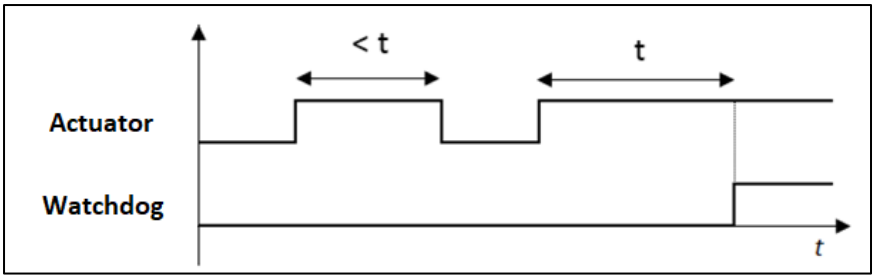


Figure 5.8 - Watchdog signaling a malfunction on an Actuator (taken from previous work)

Table 5.7 shows Timeout Monitor's variables:

Table 5.7 - Timeout Monitor's variables

Timeout Monitor			
Name	Type	I/O	Description
IN	BOOL	Input	Actuator output signal
Ref_time_off	INT		Off time limit (milliseconds)
Ref_time_on	INT		On time limit (milliseconds)
Alarm	BOOL	Output	Alarm signal

Figure 5.9 shows a representation of the "Timeout monitor" function block created in the previous work.

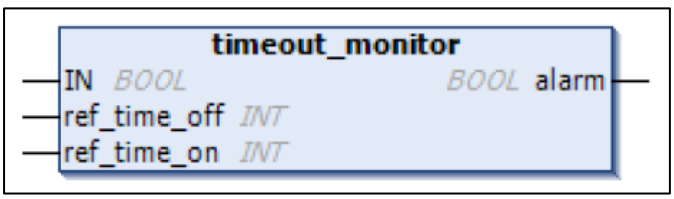


Figure 5.9 - "Timeout monitor" function block created in the previous work

5.2.2 Scenario's Description

This section will study a scenario which featured a complex automated system meant to transport items on pallets. Figure 5.10 shows the scenario designed by the author of the previous work:

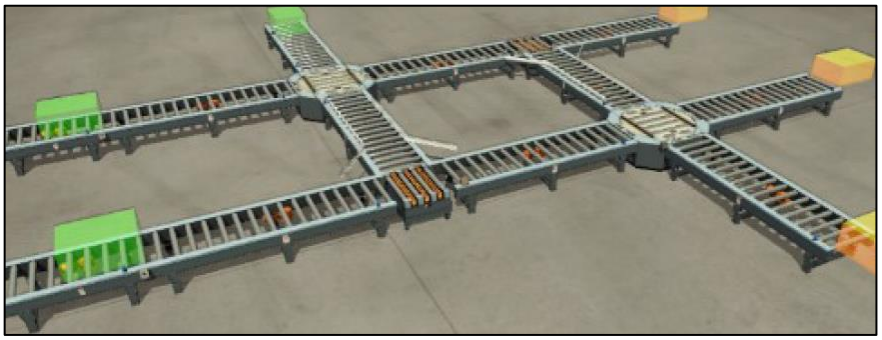


Figure 5.10 - Complex automated system designed by the author of the previous work using Factory IO

Table 5.8 describes the scenario shown in Figure 5.10:

Table 5.8 - Description of the complex automated system scenario

Pallet Transport	
Scenario	System with two Turntables and two Chain Transfer Tables connected by Roller Conveyors. The two closest tables to the removers have a button responsible for the definition of their paths. Figure 5.11 shows the position of the button on one of the tables. If the button is triggered, parts are sent to the right of the table, otherwise parts are sent forward. These trajectories are demonstrated in Figure 5.12, where **green arrows** indicate the path when the button is triggered and **black arrows** indicate the path when the button is not triggered.
Aims	Complex system automatization. Verify its ease of implementation.
Components	13 Roller Conveyors – 10 large ones and 3 small ones for the emitters; 3 Emitters and 3 Removers; 2 Chain Transfer Tables; 2 Turntables; 2 Buttons – one in each of the closest tables to the removers; 13 Sensors – one at the end of each conveyor.
POUs	Emitter; Remover; Belt Conveyor Analog (also applicable to Roller Conveyors); Chain Transfer Table; Turntable.

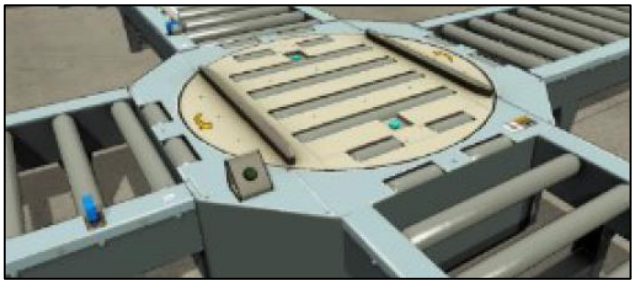


Figure 5.11 - Turntable detail: button

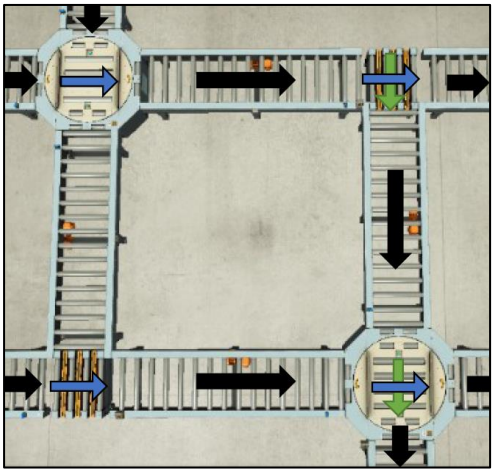


Figure 5.12 - System's possible trajectories

5.3 Improving the Studied Scenario using OOP

In order to improve the scenario, some components were added and some changes were applied to the function blocks defined in the previous work. The next section will thoroughly explain those changes.

The scenario suffered some physical changes in order to remove some unnecessary components that were increasing the simulation's duration. Some other components were introduced in order to implement some additional requirements.

5.3.1 Introduction of New Components

In order to implement some additional features to show the full potential of Object-Oriented Programming, some changes were made in the scenario:

- Three conveyors were removed;
- Every emitter has a Box Identification System attached to it, which is the same system that was used in section 4.2.2. Figure 5.13 recalls that system:

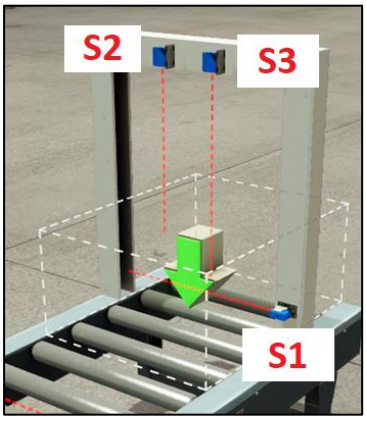


Figure 5.13 - Box Identification System attached to an Emitter

The Box Identification System features 3 sensors. Table 5.9 shows the different combinations of their signals that identify different boxes:

Table 5.9 - How the Box Identification System works

Box Identification System			
Box	Sensor 1	Sensor 2	Sensor 3
Small	TRUE	FALSE	FALSE
Medium	TRUE	FALSE	TRUE
Large	TRUE	TRUE	TRUE

- The Removers were subbed by Low Chute Conveyors. Figure 5.14 shows a Low Chute Conveyor:

Figure 5.14 - Low Chute Conveyor

- The Chain Transfer Table now features four additional sensors to assure the correct positioning of the pallets. Figure 5.15 shows the sensors' positions:

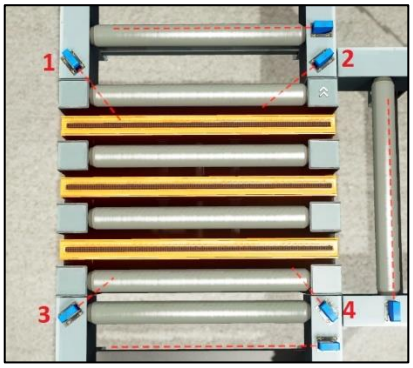


Figure 5.15 - Chain Transfer Table's four new sensors

- The System now sorts parts based on their size instead of buttons.

5.3.2 POUs that Control the New Scenario

The POUs that control each component of the scenario have also been subjected to some changes.

5.3.2.1 Conveyor with FIFO

Roller Conveyors are now controlled by the Conveyor with FIFO function block that was described in section 4.2.2. It was designed to control a belt conveyor but it also works for roller conveyors.

5.3.2.2 Tables

The “Tables” function block is a new function block that was created to deal with the common variables and implementation of the Chain Transfer Table and Turntable.

In section 5.2.1.3, Table 5.6 defined the variables that both tables had in common. The Chain Transfer Table and the Turntable **are** Tables. If both tables have common code implementation, Inputs, Outputs and Internal variables, then it means that **inheritance** is applicable.

Table 5.10 describes the new “Tables” function block that will be extended by the “Chain Transfer Table” and “Turntable” function blocks.

Table 5.10 - Description of the new "Tables" function block

Tables			
Description	Both tables share some common variables and implementation. This function block takes care of everything they have in common		
Control variables			
I/O	Type	Name	Description
Input	ARRAY OF BOOL	***a_bBusy_in***	Component’s availability in each direction: TRUE – Component is busy; FALSE – Component is available.
		loadSide	Signals if the conveyor is supposed to load parts.
		unloadSide	Signals if the conveyor is supposed to unload parts.
	INT	***iDestination***	Pallet’s destination: 1: Back; 2: Left; 3: Front; 4: Right.
	ARRAY OF INT	***a_iPartTypeIn***	ID of the part that is entering the table.
	BOOL	***bSensorBack*** ***bSensorLeft*** ***bSensorFront*** ***bSensorRight***	Sensors attached to each side of the table.
Output	ARRAY OF BOOL	***a_bBusy_out***	Table’s availability in each direction: TRUE – Table is busy; FALSE – Table is available.
	INT	***iPartTypeOut***	ID of the part that is leaving the table.

The extensions – the Chain Transfer Table and Turntable – only need to implement the code that takes care of their functional differences. For instance, the turntable FB only requires the addition of the variables and implementation defined in Table 5.5.

The “Tables” function block opens up a great possibility: The Chain Transfer Table and the Turntable can simply be treated as Tables. Both components perform the same task, which is to transfer a part between 4 possible destinations. The data that might be important for other components is available in the “Tables” FB.

So, an array of pointers to the “Tables” function block was created in order to identify all the tables of the system as the same type of component. The roller conveyors are connected to Table 1 and Table 2 instead of being connected to a specific type of table. This is very important for the application of future changes.

Figure 5.16 shows how a conveyor sees each table – not as a Chain Transfer Table or as a Turntable, but as a simple instance of a Table:

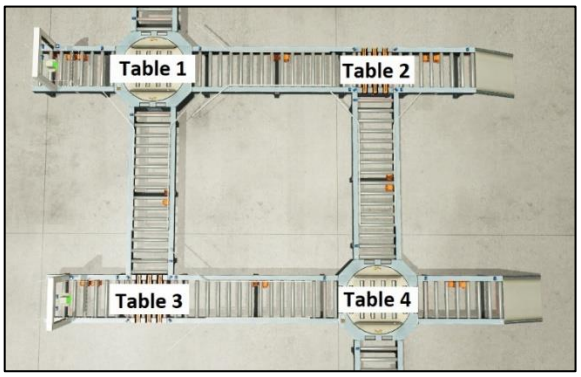


Figure 5.16 - Designation of each table

Figure 5.17 shows an array of pointers to tables in the main program. The pointers select the address of a specific table. Figure 5.18 shows a Roller conveyor communicating with the tables it connects:

```
Table: ARRAY [1..4] OF POINTER TO Tables:= [    ADR(TurnTbl[1]),
                                                ADR(CT_Table[1]),
                                                ADR(CT_Table[2]),
                                                ADR(TurnTbl[2])];
```

Figure 5.17 - “Table” Array in the main program

```
50  RollerConveyor[4](  partTypeIn:=Table[1]^.iPartTypeOut,
51                      entrySensor:=GVL.In_14,
52                      exitSensor:=GVL.In_16,
53                      nextCompBusy:=Table[2]^.a_bBusy[1],
54                      maxParts:=3,
```

Figure 5.18 - Roller Conveyor communicating with the tables it connects in the main program

If a Turntable is malfunctioning and the only the only available replacement is a Chain Transfer Table, then the programmer only has to add a new instantiation of the “Chain Transfer Table” FB to the main program and attach it to the correct position of the array, and it will automatically be connected to the corresponding conveyors.

Not only does this reduce the time required to apply changes as well as it reduces error proneness.

5.3.2.3 Timeout Monitor

Using Object-Oriented Programming, the “Timeout Monitor” FB that was described in section 5.2.1.4 can be used to perform tasks on the function blocks that call it.

Before explaining how this works, the concept of **Abstract Class** (or Abstract function block) must be introduced: it is a POU which may have standard empty methods, properties and variables. It works almost like an interface, but it may also have standard variables.

If all actuators of a system extend the same abstract class, then all of them may have something in common, which could be a “stop()” method.

The “Timeout Monitor” function block could be used to not only detect a malfunction in an actuator, but also **directly** switch it off without requiring additional implementation in the actuator’s function block, by running the “stop()” method of the function that is calling it. This can be done without requiring multiple attachments, the “Timeout” function block only needs to be called by the actuator’s function block.

This feature is possible using inheritance and pointers, but it requires 3 steps:

1. The instantiation of the “Timeout Monitor” function block in each of the actuators’ function blocks must be followed by the **THIS** keyword. Figure 5.19 shows that is done:

```
22      rollTimeout : Timeout_Monitor(THIS);
```

Figure 5.19 - Instantiation of the "Timeout Monitor" FB in the "Conveyor" FB

2. The “Timeout Monitor” function block must have a pointer capable of pointing to whoever is calling it. Figure 5.20 shows how to do it:

```
16      Actuator: POINTER TO ActAbsClass;
```

Figure 5.20 - Instantiation of a pointer to the actuator abstract class in the "Timeout Monitor" FB

Figure 5.21 shows the pointer executing some actuator’s “stop()” method:

```
13      Actuator^.Stop();
```

Figure 5.21 - Implementation in the body of the "Timeout Monitor" FB

3. That pointer must be initialized in the “FB_Init()” method of the “Timeout Monitor” FB. Figure 5.22 shows how to create that method. The method is always implicitly available, which means that it must include the “bInitRetains” and “bInCopyCode” boolean variables in order to avoid errors.

```
1  METHOD FB_Init
2  VAR_INPUT
3      bInitRetains: BOOL;
4      bInCopyCode: BOOL;
5      pInit: POINTER TO ActAbsClass;
6  END_VAR
7

1  Actuator := pInit;
```

Figure 5.22 - "FB_Init()" method of the "Timeout Monitor" FB

So, the “Timeout Monitor” FB turns off an actuator upon the detection of a malfunction in the following way:

- An actuator calls the “Timeout Monitor” FB to supervise a variable;
- The “Timeout Monitor” FB detects an anomaly and executes the “stop()” method of the pointer;
- The pointer hasn’t been manually assigned to any address;
- The pointer is able to point to whichever FB is calling it due to the THIS keyword.

This unleashes a very powerful feature: it is possible to have generic function blocks applicable to every actuator. Programmers do not need to manually attach the function blocks as it is done automatically, greatly reducing error proneness.

It also unleashes another very powerful feature, related with the use of function blocks as states. The next section explores addresses this subject.

5.3.2.4 Chain Transfer Table and Turntable

Using the approach that was exposed in the last section, it is possible to use function blocks to define the state of a component.

Figure 5.23 recalls the Chain Transfer Table's behavioral GRAFCET:

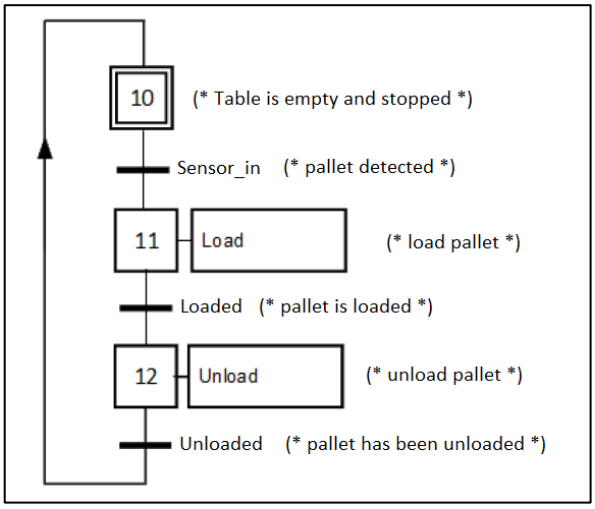


Figure 5.23 - Chain Transfer Table's behavioral GRAFCET

Different tasks are executed in each step of the GRAFCET, which means that separating the code could be a great advantage.

Using the approach that was exposed in the last section along with **interfaces** and arrays, the table can call a different function block to perform changes on itself depending on the step it is at. After finishing its task, the function block increments the step of the system so that another function block can perform another task.

Four state function blocks can be created for the Chain Transfer Table:

- "Idle" FB, which corresponds to the initial step, which is supposed to wait for a pallet to arrive;
- "Load" FB, which corresponds to the second step, which loads a part to the table;
- "Unload" FB, which corresponds to the third step, which unloads a part from the table.
- "Reset Step" FB, which is required to return to the initial step since the previous FB's only increment the step.

These four function blocks fit the GRAFCET defined in Figure 5.23 almost perfectly, the only difference is the "Reset Step" FB.

Figure 5.24 shows a UML representation of the state function blocks and their dependencies.

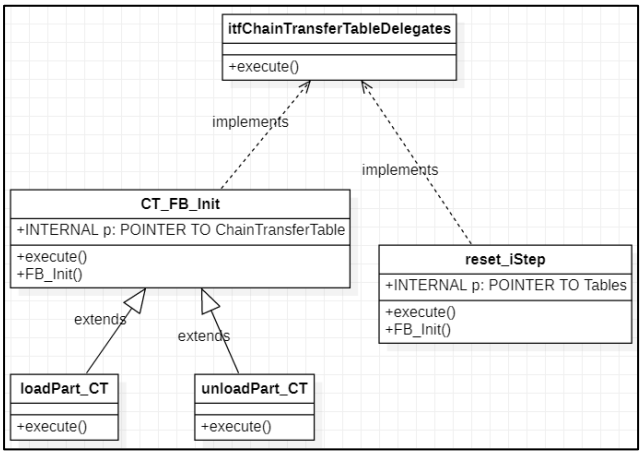


Figure 5.24 - UML representation of the Chain Transfer Table's state function blocks and their dependencies

The "CT_FB_Init" FB implements the pointer to the "Chain Transfer Table" function block and the "FB_Init()" method. The other function blocks extend it to avoid having to copy and paste. Since it's body and "execute()" method remains empty, it was used for the "Do nothing" instantiation.

The code was, thus, separated in 4 function blocks. The "itfChainTransferTableDelegates" which defines an "execute()" method:

- All function blocks implement the same interface;
- The main function block defines an array of that interface;
- The array organizes the function blocks;
- The body code implementation of the main function block orders the execution of the "execute" method depending on the step. Table 5.11 shows which FB is selected depending on the task:

Table 5.11 - Function block that is selected to perform a task depending on the step

iStep	Function block that the interface selects to run the "execute()" method
10	***Idle***
11	***Load***
12	***Unload***
13	***Reset Step***

Table 5.12 describes the additional functionalities that were added in relation to the function block described in Table 5.3:

Table 5.12 - Description of the new "Chain Transfer Table" function block

Chain Transfer Table			
Description	The Chain Transfer Table's behavioral GRAFCET has only 3 steps. It is, thus, easy to divide the code.		
Control variables			
Variable	Type	Name	Description
Property	BOOL	***p_bRollBack***	The state function blocks cannot directly access the outputs of the system, only its inputs and internal variables, properties and methods, hence the need for these properties.
		p_bRollLeft	
		p_bRollFront	
		p_bRollRight	
Internal	INT	***iStep***	Step of the GRAFCET
	CT_FB_Init	***Idle***	During the initial step, the table "does nothing" it just waits until it has to transfer a part.
	loadPart_CT	***Load***	As soon as a part arrives at one of the entrances of the table, it starts loading the part. Figure 5.27 shows part of its implementation code and how it applies changes to the main function block.
	unloadPart_CT	***Unload***	As soon as the part is loaded, the table unloads it to the desired destination.
	reset_iStep	***Reset Step***	Returns to the initial step.
	ARRAY OF Interface*	***a_state***	Array that organizes the function blocks that are going to perform tasks depending on the step. Figure 5.25 shows the instantiation of the "a_state" array of the Chain Transfer Table function block. Figure 5.26 shows the array's method being run depending on the which step the table is at.

*the interface is "itfChainTransferTableDelegates".

```
a_state: ARRAY [0..3] OF itfChainTransferTableDelegates := [do_nothing, load, unload, reset_step];
```

Figure 5.25 - Instantiation of the "a_state" interface array in the Chain Transfer Table FB

```
14  a_state[iStep].execute();
```

Figure 5.26 - Main function block's body: execution of the interface's method

```
1  CASE p^.iCurrentPosition OF
2      1: p^.p_bRollFront := TRUE;
3      2: p^.p_bRollRight := TRUE;
4      3: p^.p_bRollBack := TRUE;
5      4: p^.p_bRollLeft := TRUE;
6  END_CASE
```

Figure 5.27 - Part of the implementation of Load's "execute()" method

The same approach was applied to the “Turntable” FB. Figure 5.28 recalls the Turntable’s behavioral GRAFCET:

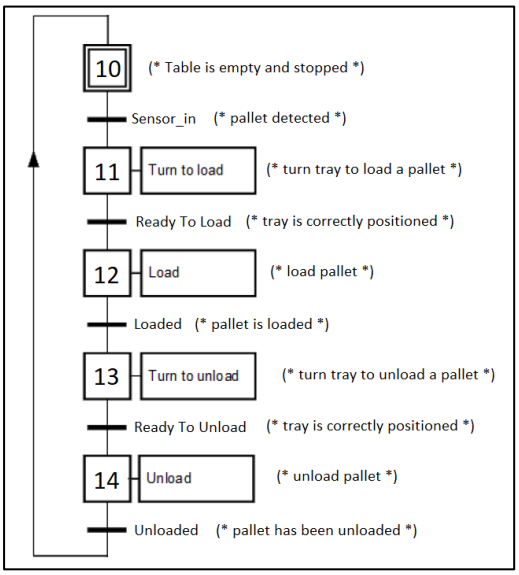


Figure 5.28 - Turntable's behavioral GRAFCET

Six state function blocks can be created for the Chain Transfer Table:

- “Do Nothing” FB, which corresponds to the initial step, which is supposed to wait for a pallet to arrive;
- “Turn to Load”, which corresponds to the second step, which turns to the side where a part is waiting to be loaded;
- “Load” FB, which corresponds to the third step, which loads a part to the table;
- “Unload” FB, which corresponds to the fourth step, which unloads a part from the table.
- “Turn to Unload”, which corresponds to the fifth step, which turns to the side where the part is supposed to be unloaded;
- “Reset Step” FB, which is required to return to the initial step since the previous FB’s only increment the step.

These four function blocks fit the GRAFCET defined in Figure 5.28 almost perfectly, the only difference is the “Reset Step” FB.

Figure 5.29 shows a UML representation of the state function blocks and their dependencies:

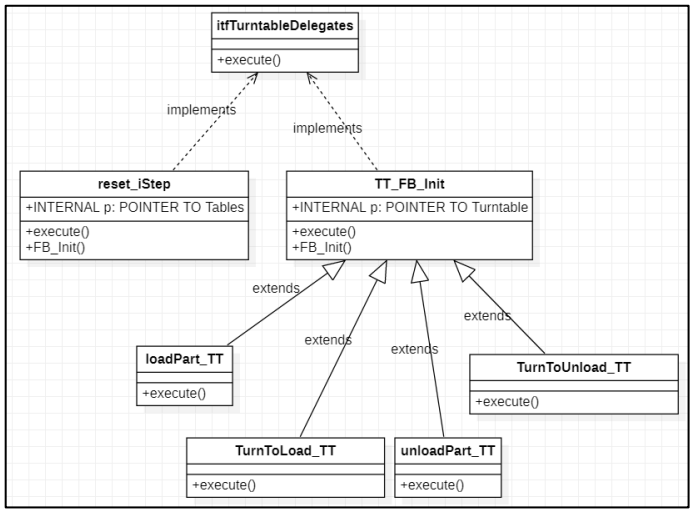


Figure 5.29 - UML representation of the Turntable's state function blocks and their dependencies

The "TT_FB_Init" FB implements the pointer to the "Turntable" function block and the "FB_Init()" method. The other function blocks extend it to avoid having to copy and paste. Since it's body and "execute()" method remains empty, it was used for the "Do nothing" instantiation.

Table 5.13 shows which function block is selected depending on the step:

Table 5.13 - Function block that is selected to perform a task depending on the step

iStep	Function block that the interface selects to run the "execute()" method
10	***Idle***
11	***TurnToLoad***
12	***Load***
13	***TurnToUnload***
14	***Unload***
15	***Reset Step***

Table 5.14 describes the new “Turntable” function block:

Table 5.14 - Description of the new "Turntable" function block

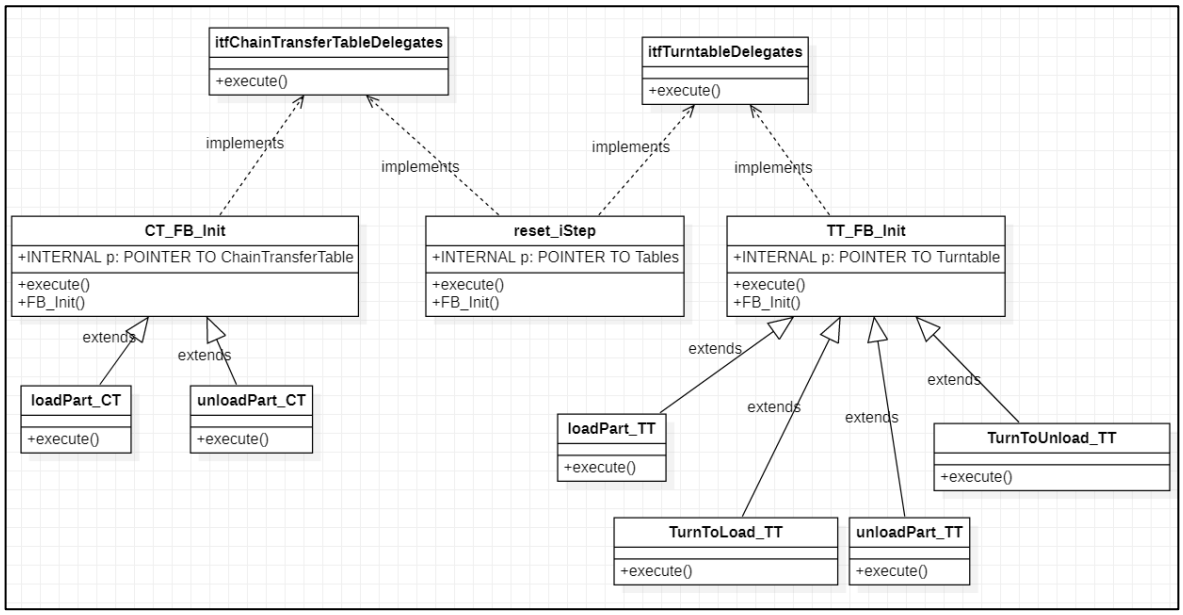
Turntable			
Description	The Turntable’s behavioral GRAFCET has 5 steps. The code was separated in 5 function blocks.		
Control variables			
Variable	Type	Name	Description
Property	BOOL	***p_bRollBack***	The state function blocks cannot directly access the outputs of the system, only its inputs and internal variables, properties and methods, hence the need for these properties.
		p_bRollFront	
		p_bTurn	
Internal	INT	***iStep***	Step of the GRAFCET
	TT_FB_Init	***Idle***	During the initial step, the table “does nothing” it just waits until it has to transfer a part.
	TurnToLoad_TT	***TurnToLoad***	The tray of the table turns to the position where an item is waiting to be loaded.
	loadPart_TT	***Load***	The table loads the part.
	TurnToUnload_TT	***TurnToUnload***	After a successful loading, the tray of the table turns to the desired destination.
	unloadPart_TT	***Unload***	The table unloads the part.
	reset_iStep	***Reset Step***	Returns to the initial step.
	ARRAY OF Interface*	***a_state***	Array that organizes the function blocks that are going to perform tasks depending on the step.

*the interface is “itfTurntableDelegates”.

Both systems may share the “a_state” variables and some states with the same name. However, they have different implementations and apply changes on different systems. The systems should use different interfaces to avoid having the Turntable’s states performing activities on the Chain Transfer Table and vice-versa.

Table 5.15 shows the full representation of the system's state function blocks and their dependencies:

Table 5.15 - UML representation of the system's state function blocks and their dependencies



The creation of function blocks as states of a component is very important for the implementation of future changes and troubleshooting. If the table is malfunctioning while loading a part, then the troubleshooter knows exactly where to search for the bug.

On the other hand, the table could have different functionalities in other systems. Having modular state function blocks that may be ordered in various different ways, depending on the goal of the system is one of the biggest advantages that OOP brings.

Inserting an additional a step to the component is also very easy, the programmer can just throw the new function into the middle of the "a_state" array and if will be automatically added.

The programmer could also have various arrays to define different "modes of operation": the component may work differently depending on the selected array.

5.3.3 New Scenario's Description

Table 5.16 describes the new scenario:

Table 5.16 - Description of the new scenario

Pallet Transport	
Scenario	System that transports boxes on top of pallets using 2 Turntables and 2 Chain Transfer Tables connected by roller conveyors. Roller Conveyors are controlled by the "Conveyor with FIFO" function block defined in section 4.2.2. Boxes are identified by their size (by an identification system) as they're inserted in the system. Each table is connected to a function that defines the path that a box must take in order to reach the desired destination, according to its type. Tables load whichever part arrives first at one of the entrances. Figure 5.30 shows where boxes are supposed to be dropped. Communication between roller conveyors and tables is standardized, which means that both tables are treated as the same component.
Aims	Complex system automatization. Verify its ease of implementation.
Components	10 Roller Conveyors – 4 large ones and 6 small ones; 3 Emitters and 3 Low Chute Conveyors; 2 Chain Transfer Tables; 2 Turntables; 34 Sensors – 9 for the identification systems, 14 for the Chain Transfer Tables, 8 for the Turntables, and 3 for the exit conveyors.
POUs	Conveyor with FIFO; Tables; Chain Transfer Table; Turntable; Timeout Monitor.
Used OOP features	Inheritance; Interfaces; Methods; Delegation; Encapsulation.

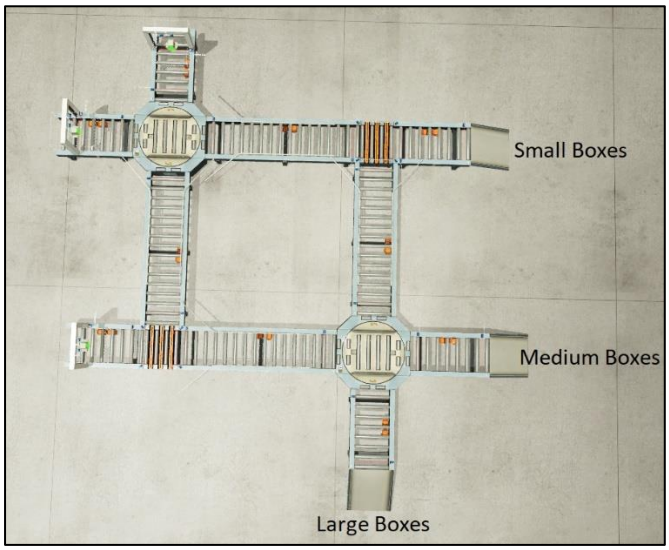


Figure 5.30 - How the system sorts the boxes

5.4 Personal overall analysis

The scenario that was studied in this chapter was greatly improved. The previous work created standard function blocks to control each individual component, which would then be directly instantiated in a program. However, attaching them to one another can prove to be a difficult task.

The creation of standardized interfaces for objects to communicate with one another, therefore minimizing the effort required to attach them and the possibility of creating modular function blocks that can easily be attached to those standardized interfaces which may already be connected to a greater set of components is an incredibly powerful advantage that enhances systems' configurability, eases troubleshooting, reduces error proneness as data management is done automatically, resulting in lower system downtimes and, consequently, better performances and more money.

Betting on modularity, configurability and customizability, supported by a strong and well-defined standardization of communication protocols between the components of a large system or factory saves lots of time in the long run, as OOP provides easily extendable and customizable PLC projects, that can sometimes be changed without directly accessing the code.

5.5 Concluding remarks

This chapter compared solutions created with and without using Object-Oriented Programming, and how this programming approach could improve the scenarios exposed in the previous work.

The next chapter addresses the final conclusions of this dissertation, and suggestions of future work.

6 Conclusions and Future Work

In the scope of this work, multiple scenarios were developed and studied in order to evaluate how Object-Oriented Programming can influence the PLC industry, which advantages and improvements it brings and its future potential in a world that increasingly demands for simple, intelligent and modular solutions.

Object-Oriented PLC Programming will certainly play a very important role in the future of the automation industry. It may take some time to grasp some of the complex concepts that it introduces, but in the end, its advantages are clear.

Object-Oriented PLC Programming sees its biggest advantages come to live when creating programming standards in one's company. Being able to standardize the way a machine works and communicates with other machines could save a lot of time in the long run. This means that OOP becomes very powerful in companies which projects often use a large portion of the same code over and over, and only small portions of custom code for an application. However, it isn't suitable for the creation of multiple, smaller programs which may not have any relation between them. OOP's way of dealing with data accessing is also much more powerful and safer than any other PLC programming approach, which is a great advantage.

Even though OOP is a much more powerful programming tool than classical PLC programming approaches, it isn't a "super tool" that can magically solve all the problems that currently exist in the industry. Ladder logic is still and will remain the most used PLC programming language in the coming years, for lots of reasons. Ladder logic and structured text, and mainly OOP, should complement each other, as they're applicable to different sets of applications due to the advantages that each of them provides.

After short conversations with lots of automation technicians and engineers from around the world, it was possible to understand a few things about the current state of the industry and why OOP won't overtake ladder logic as the main PLC programming language in the years to come.

1. Most systems are built with poor diagnostic tools:

As stated in the previous paragraph, OOP is a very powerful tool when used correctly. If programmers don't install proper diagnostic tools, it becomes very hard for technicians to debug and troubleshoot. It doesn't matter if the code is clean, well-written and easy to understand, most of the times, the problem is accessing the code.

However, as the industry evolves, and mainly with the arrival of Industry 4.0 and IoT, it is very important that systems are built with strong diagnostic and auto-diagnostic tools (and even auto-repair tools), which can save incredible amounts of time in the long run. OOP allows these strong resources through the use of its high-level programming language, which may offer possibilities that classical PLC programming approaches cannot.

2. Some of OOP's concepts are applicable to PLC programming, others aren't:
Some of OOP's core concepts have been rejected by the industry, such as inheritance, mainly because there are alternatives that can be used to achieve the same results. Many engineers claim that it excessively spreads the code and that having to hunt for the code implementation complicates troubleshooting and debugging.

However, many engineers seem to be dealing with bad implementations of OOP features, which was a very important subject that was addressed in chapter 4: these features aim to keep programs simple, not to add more complexity. If debuggers have to hunt for code, the program is either poorly documented or it has an excess of dependencies between function blocks.

This also corroborates another fact that was stated in chapter 4: OOP is not easy to learn and understand, it requires a lot of studying and practicing. Programmers need to know where and how to apply OOP's features, otherwise it can result in code that is faulty, poorly-documented and hard to understand.

Concepts like inheritance can only be used in really specific scenarios such as the ones that were exposed in this dissertation. Implementing these concepts in situations where they aren't applicable just because they're powerful concepts is often a bad practice.

3. Resistance to change:
Custom PLC software is commissioned and paid for by owners of large industrial facilities, who don't take chances against stable industries, such as the PLC industry.

Programming languages in the PLC industry haven't changed a lot in 30 years. It is possible to take code from a 30-year-old PLC onto a new processor and get it running without little effort. Machines usually run for long periods of time (10, 20, 30 years) before being updated. Since machines have to be maintained during their whole lifespan, the possibility of having new machines running the same languages as old machines is very important.

Also, Object-oriented programming isn't fully supported by a lot of PLCs. Some providers don't allow user defined types to be changed during runtime, which could become a very expensive problem: Shutting down a machine because the program needs to be recompiled could cost a lot of money and that is a risk that business owners are not willing to take.

4. Object-oriented programming isn't very suitable for the creation of safety logic:
Safety logic should be as simple as possible, and additional complexity adds more room for error and makes the validation process harder. Classical PLC programming approaches will remain the most used approach in this area.

Object-oriented PLC programming is very powerful but it there is still a lot of research to be done. A proposal for future work is the creation of highly customizable scenarios that can be altered through an HMI instead of having to access the source code to apply changes.

The creation of a system that can be almost intuitively configured by someone who doesn't know how to program PLCs, mainly without having to access someone else's code (which can be hard to understand even if well documented) can be a great demonstration of OOP's power.

References

[1] Kaloyan. (2018, 18/07/2019). *PLC (Programmable Logic Controller)*. Available: https://cyberx-labs.com/glossary/plc-programmable-logic-controller/

[2] B. Lydon. (2015, 18/07/2019). *Stimulus for New Automation Architecture*. Available: https://www.automation.com/automation-news/article/stimulus-for-new-automation-architecture

[3] L. L. World. (2019, 29/07/2019). *Relay Logic Vs Ladder Logic*. Available: https://ladderlogicworld.com/relay-logic-vs-ladder-logic/

[4] J. Reaves. (2018, 15/09/2019). *Comparing ladder logic and object-oriented programming*. Available: https://www.controleng.com/articles/comparing-ladder-logic-and-object-oriented-programming/

[5] T. Walter. (2007, 13/03/2019). *Ladder logic: Strengths, weaknesses*. Available: https://www.controleng.com/articles/ladder-logic-strengths-weaknesses/

[6] T. R. Kuphaldt, "Ladder Logic Arithmetic Instructions," ed: Creative Commons Attribution 4.0 License, 2017.

[7] G. Pratt. (2019, 27/01/2020) Leveraging OOIP, Part 2: Abstraction, nesting, interfaces. *Control Engineering*. 32-35.

[8] S. Thompson. (1996, 27/01/2019). *Haskell: The Craft of Functional Programming* Available: https://www.cs.kent.ac.uk/people/staff/sjt/Haskell_craft/preface.html

[9] K. Eliason. (2013, 17/07/2019). *Difference Between Object-oriented Programming and Procedural Programming Languages*. Available: https://neonbrand.com/website-design/procedural-programming-vs-object-oriented-programming-a-review/

[10] T. Janssen. (2017). *OOP Concepts for Beginners: What is Polymorphism*. Available: https://stackify.com/oop-concept-polymorphism/

[11] E. Elliott. (2018). *The Forgotten History of OOP*. Available: https://medium.com/javascript-scene/the-forgotten-history-of-oop-88d71b9b2d9f

[12] B. Babcock. (1999, 17/07/2019). *IEC-1131 - The First Universal Process Control Language*. Available: https://www.automation.com/library/articles-white-papers/process-control-process-monitoring/iec-1131-the-first-universal-process-control-language

[13] International Electrotechnical Comission, *IEC 61131 Programmable controllers - Part 3: Programming Languages*. 2013.

[14] M. Hamsho, "PLC Object Oriented Programming: Advanced Infrastructure," ed. Udemy, 2018.

[15] S. Henneken. (2014, 19/03/2019). *IEC 61131-3: Object composition with the help of interfaces*. Available: https://stefanhenneken.wordpress.com/2014/02/18/iec-61131-3-object-composition-with-the-help-of-interfaces/

[16] S.-S. S. S. GmbH. (2020, 05/01/2020). *CODESYS Development System*. Available: https://www.codesys.com/

[17] S.-S. S. S. GmbH, "CODESYS V3.5 SP 4 Compliance Table," vol. 2017, ed: 3S-Smart Software Solutions GmbH, 2017.

[18] M. Braun, *Object-oriented programming in simotion*. [Place of publication not identified]: Publicis Mcd Verlag, Germa, 2017.
[19] R. Games. (2020, 05/01/2020). *Factory IO - User Manual*. Available: https://factoryio.com/
[20] International Electrotechnical Comission, *IEC 60848 GRAFCET specification language for sequential function charts*. 2013.
[21] L. E. C. d. Santos, "Controlo Modular e Confiável de Sistemas Flexíveis de Automação," Mechanical Engineering Dissertation, Engineering, Universidade do Porto, 2015.

All web pages were available at the dissertation's submission date (27/01/2019).