Centro de Ingeniería y
Desarrollo Industrial

CONACYT
Consejo Nacional de Ciencia y Tecnología

Hochschule Aachen
Aachen University
of Applied Sciences

Use and adoption of software design patterns for PLC based systems

THESIS

TO OBTAIN THE ACADEMIC

DEGREE OF

MASTER IN MECHATRONICS

BY:

ARMANDO RENE NARVÁEZ CONTRERAS

SANTIAGO DE QUERETARO, QRO., FEBRUARY 2017

Use and adoption of software design patterns for PLC based systems

Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where acknowledgement has been made in the text.

Armando Rene Narvaez Contreras Queretaro 02.2017

Acknowledgements

I would like to thank my thesis advisors Dr. Alfonso Gómez Espinosa and Prof. Jörg Wollert for their willing to help in this work. They consistently allowed this thesis to be my own work, but steered me in the right the direction whenever they thought I needed it.

I would also like to thank B.Sc. Sebastian Rau as the second reader of this thesis, and I am gratefully indebted to him for his very valuable comments on this thesis.

I would also like to acknowledge CONACYT and my colleagues from CIDESI in Mexico for giving me the opportunity, support and resources to study in this program.

I want to thank my parents and family for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis.

Finally, I must express my very profound gratitude to my wife Daniela and my daughter Paula for giving me light when I needed it the most. This accomplishment would not have been possible without them. Thank you.

Abstract

The international standard IEC 6113-3 specifies the syntax and semantics on which most of the Programmable Logical Controllers (PLC) are programmed as of today. In the latest update to the standard the concept of domain specific object orientation, was approved to be deployed and implemented on PLC based systems. This opens up a wide variety of possibilities from the vast and researched field of object orientation from Personal Computers (PC) based systems to be adopted to the automation industry.

One specific subject that can be adopted from the PC based systems into the PLC based systems are, the already existing software design patterns applied to industry related problems in software development of industrial controllers.

This thesis takes some of the software design patterns defined in the Information Technology (IT) field and adopts them to the automation industry, with help of Unified Modeling Language (UML) notation to present domain specific tools in software development, that can be directly applied to the software development stage in the automation industry.

Contents

List of Figures

1 Introduction

Currently, most of the industrial automation software systems are programmed following the IEC 6113-3 standard. This standard implemented new Object-Oriented (OO) extensions to its latests version giving the industrial automation systems the possibility to handle an Object-Oriented Programming (OOP) approach to its applications[1].

Some of the advantages in the use of OOP for automation systems are, better-structured program code with separation of concerns and information hiding, flexible extensibility by new types of objects (e.g., software representations of new types of drives), reuse of code for defining specialized subclasses (inheritance) and the reuse of code operating on different implementations of an interface (polymorphism)[2].

The first implementation of Object-Oriented IEC 6113-3 was realized by CoDeSys [1] and then implemented by the company Beckhoff with their Integrated Development Environment (IDE) TwinCat.

This thesis presents the adoption and implementation of software engineering Object-Oriented Programming, Unified Modeling Language and design patterns to solve recurrent problems in software development for industrial automation.

1.1 Statement of the Problem

The complexity of the current automation systems is steadily increasing at a very high pace, development cycle times are decreasing, and more and more tasks are assigned to the controller software[2]. According to the German Engineering Federation VDMA, the ratio of software has doubled in one decade from 20% to 40%. If this trend continues, software engineering will be the main activity of automation systems suppliers and developers [3]. In order to address this constantly growing complexity, new techniques and methods of the automation systems development and automation software development are required [4].

Reflecting on the trend of growing importance of software in automation, there is a great number of research projects and corresponding publications addressing various aspects of software development process in industrial automation domain. The main driving forces of these developments are life-cycle costs, dependability, and performance. The cost issues are addressed through the entire life-cycle of software, from requirements capturing to phasing the software out. The dependability related challenges focus on the methods and activities ensuring functional safety of computer systems through guaranteeing certain safety properties of the software. The performance-related developments aim at ensuring sufficient execution speed and lower memory footprint of the code. These characteristics can be interdependent, for example, certain dependability guarantees may depend on sufficient performance, and higher performance of software (achieved on account of more efficient coding or compilation) can reduce the overall costs of automation system by using lower spec hardware. All of these characteristics of computer hardware and software certainly impact on the performance and reliability of systems being automated [3].

1.2 Hypothesis

Is possible to implement software Builder, Decorator, Observer, Proxy and Singleton design patterns for PLC systems.

1.3 Objective

The main objective of this work of investigation is the following:

- Design and implement software Builder, Decorator, Observer, Proxy and Singleton design patterns for PLC based systems.

For the main objective to be done is necessary first to accomplish the following specific objectives:

- Explain the common use of each design pattern.
- Model each design pattern in Unified Modeling Language (UML) notation.
- Generate code template from the UML models.
- Propose an industrial application to be solved with the specific design pattern.
- Implement, compile and simulate the proposed industrial application using design patterns in a PLC Integrated Development Environment (IDE).

1.4 Approach

An important research method in industrial automation is to adopt developments from the general computing area. This is the case for virtually any software-related technology, e.g., component-orientation, service-orientation or model-based engineering, to mention a few. This allows the developers to take advantage of the huge investments into such technologies and rely on proven solutions rather than reinventing the wheel [3].

Design patterns are a well known element in software engineering which brings trusted solutions to common design problems[5]. Design patterns are solution approaches that describe, how typical and recurring design problems are solved. A design pattern consists of a description of the design problem, a possible solution as well as the context in which the solution is valid [6].

Design patterns make it easier to reuse successful designs and architectures. Expressing proven techniques as design patterns makes them more accessible to developers of new systems. Design patterns help you choose design alternatives that make a system reusable and avoid alternatives that compromise reusability. Design patterns can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent [6].

This thesis adopts the software engineering design patterns, UML and the new capabilities of OOP in the CODESYS IDE, to address specific situations on the software development for industrial automation. The use of UML models to produce code templates and design patterns, accelerates the software development life-cycle reducing the code footprint memory and communication dependability within the system.

The reminder of the thesis is structured as follows: In Chapter 2, the current State of the Art will be briefly introduced, including the UML notation and the basic concepts of Design patterns, as explained in the literature. In Chapter 3, the work of adoption of the previous tools is explained and examples of industrial applications related to the automation industry are given. Finally, in Chapter 4, the conclusions and outlook for this thesis are given.

2 State of the Art

2.1 IEC 61131

The International Electrotechnical Commission (IEC) is a not-for-profit, non-governmental worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides.

The IEC held its inaugural meeting on 26 June 1906, following discussions between the British IEE, the American Institute of Electrical and Electronics Engineers (IEEE) (then called AIEE), and others. It currently counts more than 130 countries. Originally located in London, the commission moved to its current headquarters in Geneva in 1948.

2.1.1 IEC 61131-3

This part of IEC 61131 specifies syntax and semantics of a unified suite of programming languages for programmable controllers. This suite consist of two textual languages, Instruction List (LI) and Structured Text (ST), and two graphical languages, Ladder Diagram (LD) and Function Block Diagram (FBD). An additional set of graphical and equivalent textual elements named Sequential Function Chart (SFC) is defined for structuring the internal organization of programmable controller programs and function blocks. Depending on the PLC vendor (and often Country because of cultural reasons), some languages are most used.

2.2 CODESYS development environment

Some programming environments have been introduced to provide OOP for industrial automation programming, see, for instance, references [7] and [8].

CODESYS V3 is developed by a medium-size vendor of software tools can be used for programming a significant number of industrial devices, it supports PLCopen XML import/export functionality that allows to interface such a tool with other environments to improve the design of OOP applications as discussed in [9]. CODESYS meets the additional requirements listed above for OOP programming tools; it extends the IEC 61131 FB to a class construct by the addition of methods, inheritance; it introduces the INTERFACE-construct for the declaration of abstract FBs with polymorphic reference semantics [10].

2.3 OOP in Control Automation

OOP has proved to be absolutely unbeatable when It comes to elegantly handling complex software-development tasks and producing flexible, reusable software components. OOP has clearly reduced the development time of new software and simplified the solution of complex software tasks [2].

Bonfè and Fantuzzi in [11] and Secchi et al. [12] performed a practical case of study with the company Tetra Pak Carton Ambient SpA. In their study, they adopted a "Top-Down" programmed controller with an OO controller approach using UML notation.

Hee Han in [13] proposed a framework of three phases for the OO design approach in order to improve PLC programming practices presenting an example of design method dividing a complete Automation Manufacturing System into 3 sub-models based in the UML models use case, class and sequence diagrams.

Pineda-Sanchez et al. [14] Developed a practical example for the packaging industry exploiting OOP in PLC programming with benefits like cost reduction, faster manipulation and reusability of code.

Witsch and Vogel [1] presented the main object-oriented extensions to the IEC-6113-3 which are implemented in CoDeys V3, based on these extensions the authors derived UML diagrams to enhance the modularity and reusability of the systems.

2.4 UML

The Unified Modeling Language (UML) has become the de facto standard modeling notation used as a graphical notation to complement software documentation [15] in software engineering. It is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components.

UML is not a development method, that means it does not tell you what to do first and what to do next or how to design your system, but it helps you to visualize your design and communicate with others. UML is controlled by the Object Management Group (OMG) and is the industry standard for graphically describing software.

There are Three prominent types of diagrams in the UML notation:

- Structure diagrams: Showcases things that must be present in the system being modeled, the structure and substructure of the system using objects, attributes, operations, and associations. They are used extensively in documenting the software architecture of software systems. Includes Class diagrams.

- Behavior diagrams: Showcases what must happen in the system being modeled, the functionality of the system from the user's point of view. They are used extensively to describe the functionality of software systems. Includes Use Case diagrams and Activity diagrams.

- Interactions diagrams: Interaction diagrams are a subset of behavior diagrams. Showcases the flow of control and data among the things in the system being modeled and the internal behavior of the system. Includes Sequence diagrams and State Machine diagrams.

In the literature, there have been several attemps [16], [17], [18], [19], to integrate IEC 61131-3 notations with UML or SysML (Systems Modeling Language, general-porpuse modeling language for systems engineering applications.) Yet, previous research on software engineering with UML shows that while using more diagrams is confusing [17], using adapted subsets of UML is beneficial, cf. [19], [20].

Witsch in [19] developed the plcML, a domain-specifically adapted UML profile for PLCs. The plcML reduces the number of notations to three, offering the class diagram for structure modeling and adapted activity and state-chart diagrams for behavior modeling.

2.4.1 Class diagram

Class diagrams show the different classes that make up a system and how they relate to each other. Class diagrams are said to be "static" diagrams because they show the classes, along with their methods and attributes as well as the static relationships between them: which classes ”know” about which classes or which classes ”are part” of another class, but do not show the method calls between them.

A class defines the attributes and the methods of a set of objects. All objects of this class (instances of this class) share the same behavior, and have the same set of attributes (each object has its own set). The term ”Type” is sometimes used instead of class, but it is important to mention that these two are not the same, and type is a more general term.

In UML, classes are represented by rectangles, with the name of the class, and can also show the attributes and operations of the class in two other ”compartments” inside the rectangle.

In UML, Attributes are shown with at least their name, and can also show their type, initial value and other properties. Attributes can also be displayed with their visibility:

- + Stands for public attributes
- # Stands for protected attributes
- - Stands for private attributes

Operations (methods) are also displayed with at least their name, and can also show their parameters and return types. Operations can, just as attributes, display their visibility:

- + Stands for public operations
- # Stands for protected operations
- - Stands for private operations

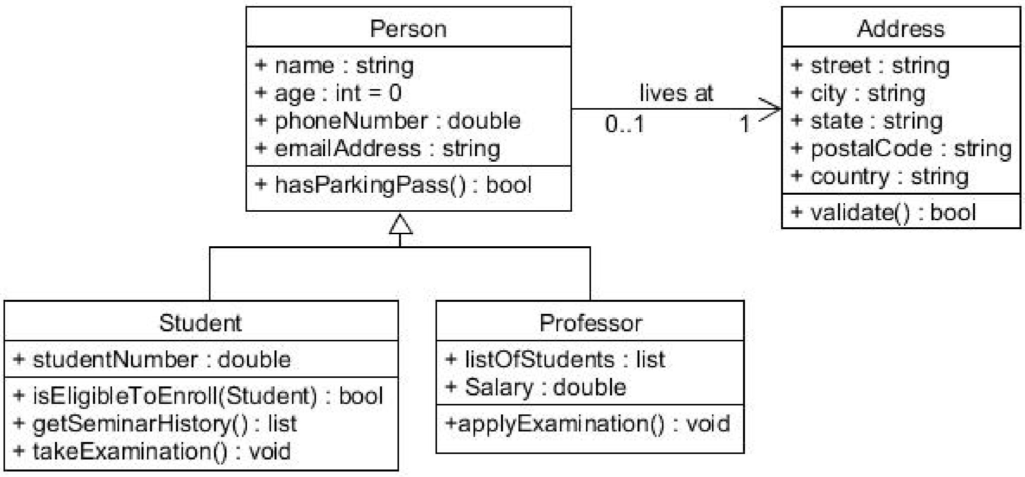


Figure 1: Simple UML class diagram

2.4.2 State diagram

State diagrams show the different states of an object during its life and the stimuli that cause the object to change its state.

State Diagrams view objects as state machines or finite automata that can be in one of a set of finite states and that can change its state via one of a finite set of stimuli. For example an object of type NetServer can be in one of following states during its life:

- Ready
- Listening
- Working
- Stopped

And the events that can cause the object to change states are

- Object is created
- Object receives message listen
- A Client requests a connection over the network
- A Client terminates a request
- The request is executed and terminated
- Object receives message stop ... etc

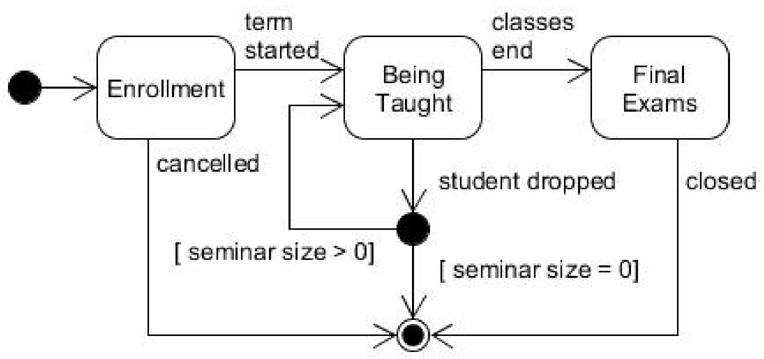


Figure 2: Simple UML state diagram

2.4.3 Activity diagram

Activity diagrams describe the sequence of activities in a system with the help of "activities". Activity diagrams are a special form of State diagrams, that only (or mostly) contains activities.

Activity diagrams are always associated to a class , an operation or a Use Case.

Activity diagrams support sequential as well as parallel activities. Parallel execution is represented via Fork/Wait icons, and for the activities running in parallel, it is not important the order in which they are carried out (they can be executed at the same time or one after the other)

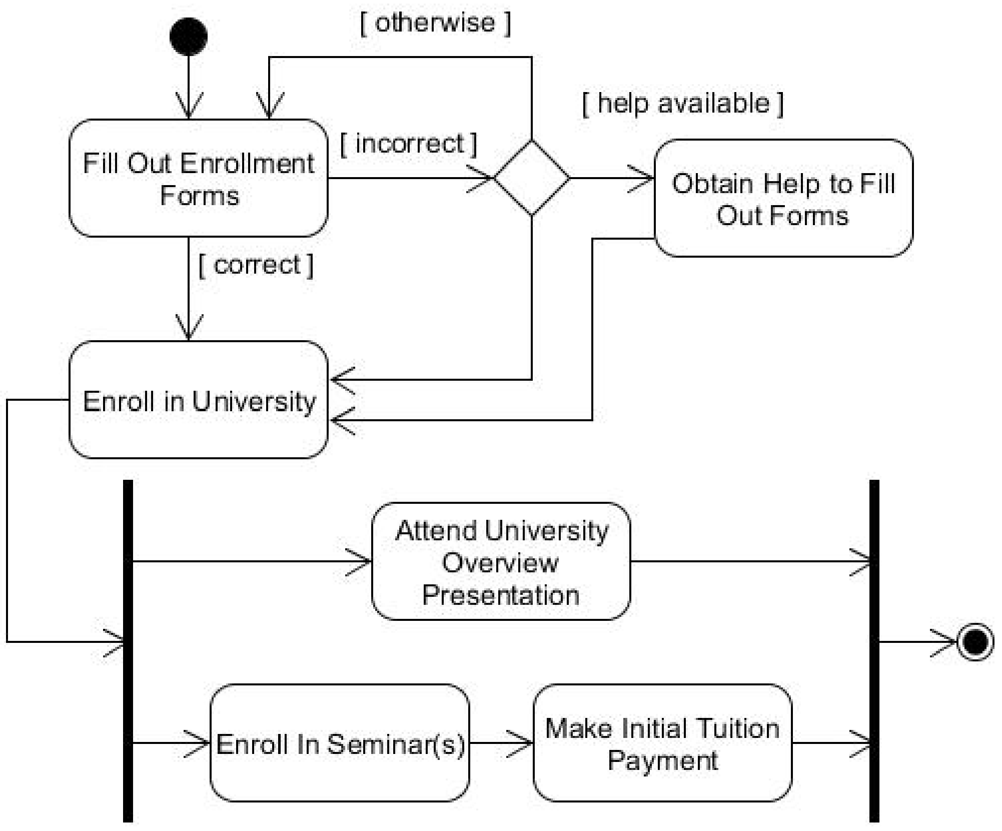

[otherwise]
[help available]
Fill Out Enrollment Forms
[incorrect]
Obtain Help to Fill Out Forms
[correct]
Enroll in University
Attend University Overview Presentation
Enroll In Seminar(s)
Make Initial Tuition Payment

Figure 3: Simple UML activity diagram

2.4.4 Sequence diagram

Sequence diagrams show the message exchange (i.e. method call) between several objects in a specific time-delimited situation. Objects are instances of classes. Sequence diagrams put special emphasis in the order and the times in which the messages to the objects are sent.

In Sequence diagrams objects are represented through vertical dashed lines, with the name of the object on the top. The time axis is also vertical, increasing downwards, so that messages are sent from one object to another in the form of arrows with the operation and parameters name.

Messages can be either synchronous, the normal type of message call where control is passed to the called object until that method has finished running, or asynchronous where control is passed back directly to the calling object. Synchronous messages have a vertical box on the side of the called object to show the flow of program control.

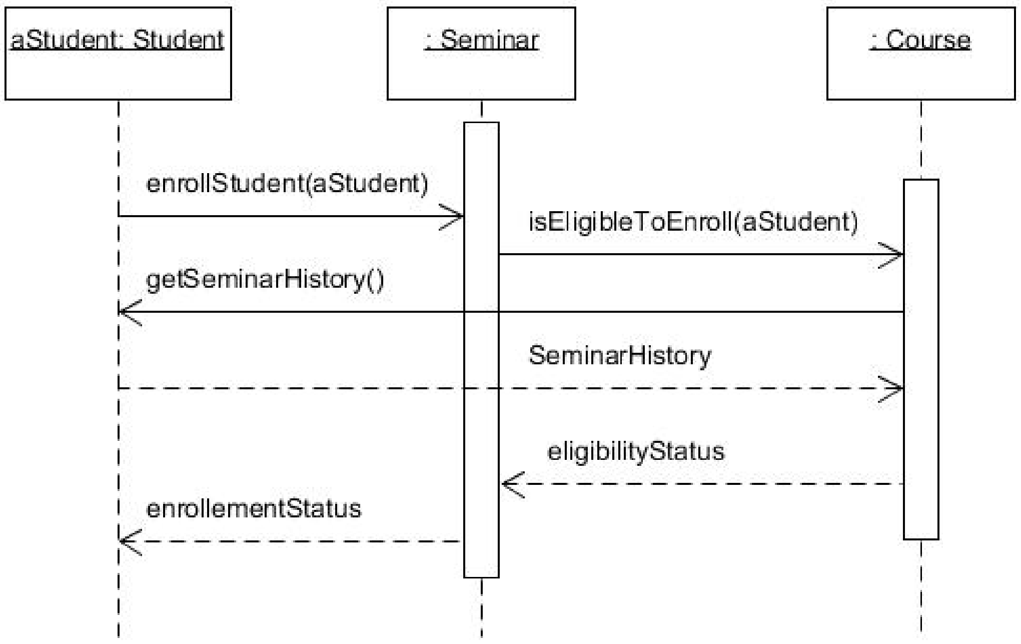


Figure 4: Simple UML sequence diagram

2.5 Design patterns concepts

As mentioned in the last chapter, Design patterns are a well known element in software engineering which brings trusted solutions to common design problems. Only software engineers name them design patterns, but it is a concept commonly found in very different disciplines. Professionals tend to give name to specific problems together with solutions widely accepted by their colleagues. When those patterns become successful, then they become jargon, and those terms will appear in the descriptions professionals make of problems, and in the conversations they carry on about them [5].

Design patterns make it easier to reuse successful designs and architectures. Expressing proven techniques as design patterns makes them more accessible to developers of new systems. Design patterns help you choose design alternatives that make a system reusable and avoid alternatives that compromise reusability. Design patterns can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent [6].

2.5.1 Builder design pattern

The Builder pattern, as described in the literature, separates the construction of a complex object from its representation so that the same construction process can create different representations.

In general, the Builder pattern is a software design pattern used to create a complex objects made up from other objects, and you want the creation of these parts to be independent of the main object.

The main known uses of the Builder pattern are:

1. When the construction process must allow different representations for the object that is constructed.

2. When the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.

The results of using the Builder pattern are:

1. It lets you vary a product's internal representation. The builder object provides the director with an abstract interface for constructing the product. The interface lets the builder hide the representation and internal structure of the product. It also hides how the product gets assembled.

2. It isolates code for construction and representation. The Builder pattern improves modularity by encapsulating the way a complex object is constructed and represented. Clients needn't know anything about the classes that define the product's internal structure. Each ConcreteBuilder contains all the code to create and assemble a particular kind of product. The code is written once; The different directors can reuse it to build product variants from the same set of parts.

3. It gives you finer control over the construction process. Instead of constructing products in one shot, the Builder pattern construct the product step by step under the director's control. Only when the product is finished does the director retrieve it from the builder.

The classic class diagram of the Builder design pattern is explained.

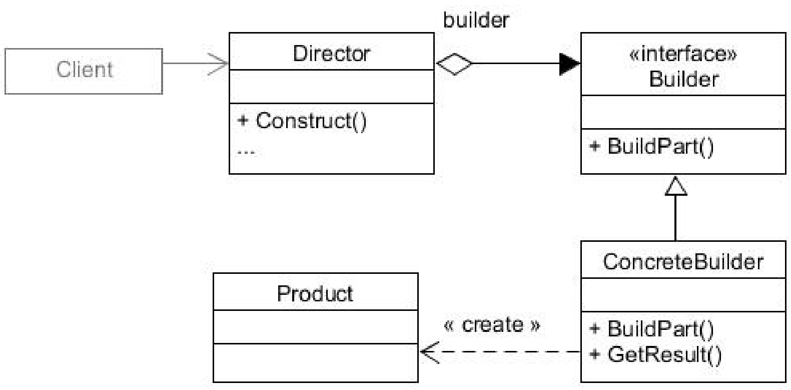


Figure 5: Builder pattern class diagram

- Builder: specifies an abstract interface for creating parts of a product object.
- ConcreteBuilder: Constructs and assembles parts of the product by implementing the builder interface, defines and keeps track of the representation it creates and provides an interface for retrieving the final product(GetResult()).
- Director: Constructs the object using the builder interface.
- Product: Represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled.

2.5.2 Decorator design pattern

The Decorator pattern, as described in the literature, attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

In general, the Decorator pattern is a software design pattern that allows you to modify an object dynamically and add functionality at run time.

The main known uses of the Decorator pattern are:

1. Add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
2. When extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination.

The result of using the Decorator pattern is:

1. More flexibility than static inheritance. The Decorator pattern provides a more flexible way to add responsibilities to objects than can be with static (multiple) inheritance. With decorators, responsibilities can be added and removed at runtime simply by attaching and detaching them. In contrast, inheritance requires creating a new class for each additional responsibility. This gives rise to many classes and increases the complexity of a system.

The classic class diagram of the Decorator design pattern is explained.

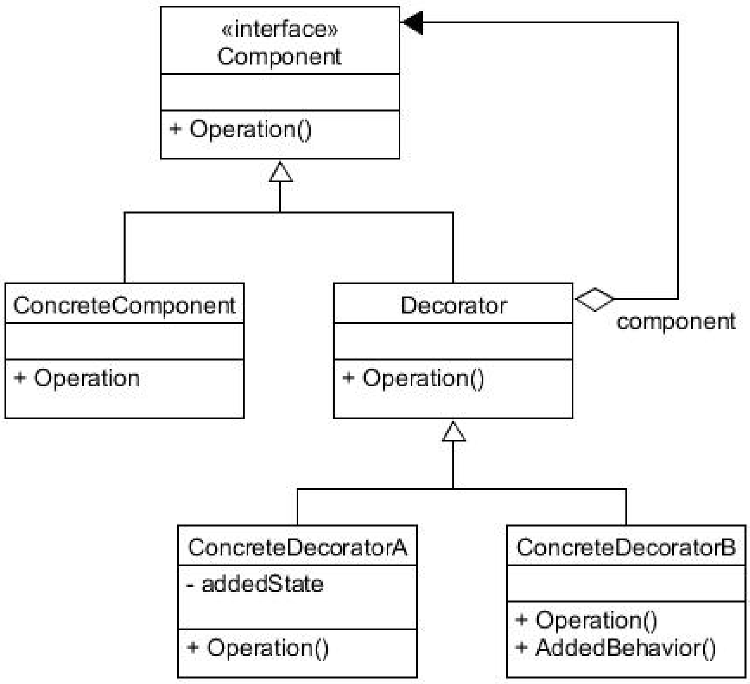


Figure 6: Decorator pattern class diagram

- Component: Defines the interface for objects that can have responsibilities added to them dynamically.

- ConcreteComponent: Defines an object to which additional responsibilities can be attached.
- Decorator: Maintains a reference to a component object and defines an interface that conforms to component's interface.
- ConcreteDecorator: Adds responsibilities and/or behavior to the component.

2.5.3 Observer design pattern

The Observer pattern, as described in the literature, defines a one-to-many dependency between objects so that when one object changes state, all its dependants are notified and updated automatically.

In general, the Observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependants, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

The main known uses of the Observer pattern are:

1. When an object should be able to notify other objects without making assumptions about who these objects are. It provides a lose coupled relationship between the subject and its observers.
2. When a change to one object requires changing others, and you don't know how many objects need to be changed.

The results of using the Observer pattern are:

1. Abstract coupling between subject and observer. All the subject knows is that it has a list of observers, each conforming to the simple interface of the abstract observer class. The subject doesn't know the concrete class of any observer. Thus the coupling between subjects and observers is abstract and minimal.
2. Support for broadcast communication. Unlike an ordinary request, the notification that a subject sends needn't specify its receiver. The notification is broadcast automatically to all interested objects that subscribed to it. The subject doesn't care how many interested objects exist; its only responsibility is to notify its observers. This gives you the freedom to add and remove observers at any time. It's up to the observer to handle or ignore a notification.

The classic class diagram of the Observer design pattern is explained.

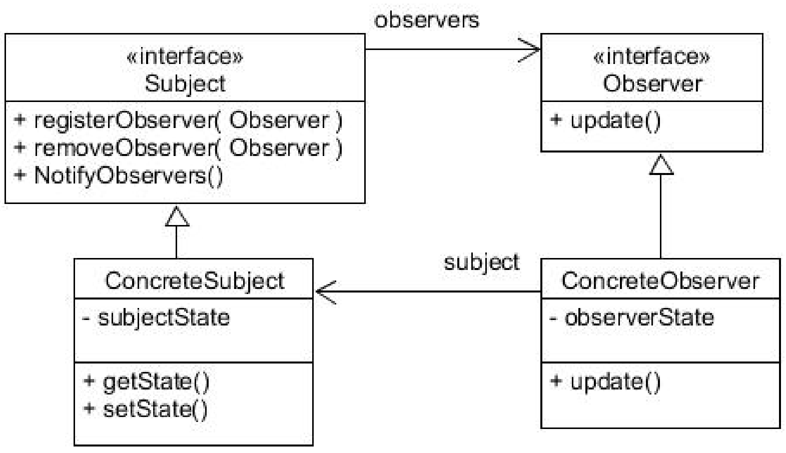


Figure 7: Observer pattern class diagram

- Subject: Provides an interface for attaching and detaching observer objects. Any number of observer objects may observe a subject.
- Observer: Defines an updating interface for objects that should be notified of changes in a subject.
- ConcreteSubject: Stores state of interest to ConcreteObserver objects. Sends a notificatio to its observers when its state changes.
- ConcreteObserver: Maintains a reference to a ConcreteSubject object. Stores state that should stay consistent with the subject's. Implements the observer updating interface to keep its state consistent with the subject's.

2.5.4 Proxy design pattern

The Proxy pattern, as described in the literature, provides a surrogate or place-holder of another object to control access to it.

In general, the Proxy pattern is a software design pattern used as an access control mechanism in charge of filtering the requests that a certain object may receive, only passing those requests with the proper access rights to the object and adding additional functionality to the specific response.

The main known uses of the Proxy pattern are:

1. Remote Proxy, a remote proxy controls access to a remote object. The remote proxy acts as a local representative for an object that lives in a different server, database, machine etc. A method call on the proxy results in the call being transferred over the communication line, invoked remotely, and the result being returned back to the proxy and then to the client.
2. Virtual Proxy, a virtual proxy controls access to a resource that is expensive to create. The virtual proxy often stalls the creation of the object until it is needed; the virtual proxy also acts as a surrogate for the object before and while it is being created. After that, the proxy delegates the requests directly to the RealSubject.
3. Protection Proxy, a protection proxy controls access to the original object. Protection proxies are useful when objects should have different access rights. The

protection proxy handles the request differently depending on the specific client or administration rights, this way the some methods won't work for the clients without the needed access rights.

4. Smart Reference, a smart reference is a replacement for a bare pointer that performs additional actions when an object is accessed.

The results of using a certain type of proxy are:

1. A remote proxy can hide the fact that an object resides in a different address space.
2. A virtual proxy can perform optimizations such as creating an object on demand.
3. Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed.

The classic class diagram of the Proxy design pattern is explained.

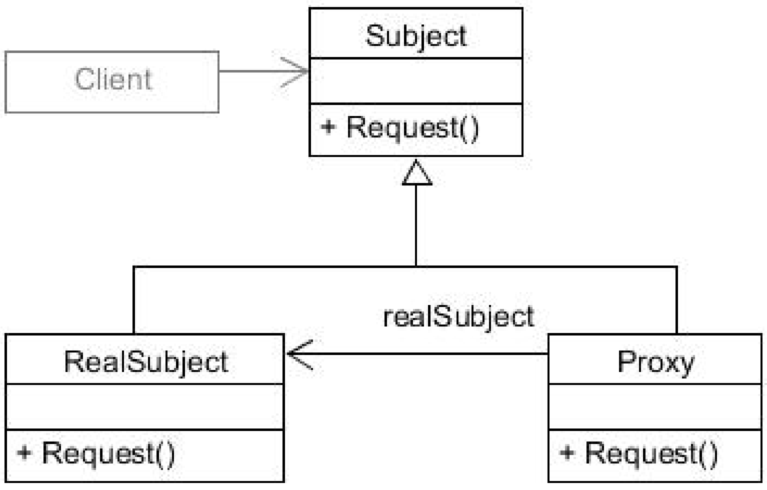


Figure 8: Proxy pattern class diagram

- Proxy: This object maintains a reference that lets the proxy access the real subject. Controls access to the real subject and may be responsible for creating and deleting it. Other responsibilities depend on the kind of proxy:
  - Remote proxies are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different server.
  - Virtual proxies may cache additional information about the real subject so that they can postpone accessing it.
  - Protection proxies check that the caller has the access permissions required to perform a request.
- Subject: This object can be either an interface or an abstract class. It defines the common interface for RealSubject and proxy so that a proxy can be used anywhere a RealSubject is expected. It should define all the possible methods available to be requested by the client.
- RealSubject: Defines the real object that the proxy represents

2.5.5 Singleton design pattern

The Singleton pattern, as described in the literature, ensures a class only has one instance, and provide a global point of access to it.

In general, the Singleton pattern is a software design pattern used to eliminate the option of instantiating more than one object.

The main known use of the Singleton pattern are:

1. When there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.

The results of using the Singleton pattern are:

1. Controlled access to sole instance. Because the singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.
2. Reduced name space. The Singleton pattern is an improvement over global variables. It avoids polluting the name space with global variables that store sole instances.
3. Permits a variable number of instances. The pattern makes it easy to change your mind and allow more than one instance of the singleton class. Moreover, you can use the same approach to control the number of instances that the application uses. Only the operation that grants access to the singleton instance needs to change.

The classic class diagram of the Singleton design pattern is explained.

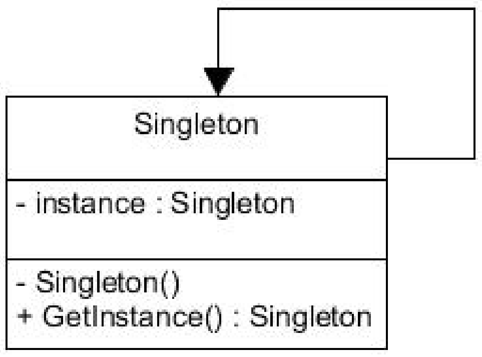


Figure 9: Singleton pattern class diagram

- Singleton: Defines a GetInstance operation that lets the clients access its unique instance. GetInstance is a class operations(Static method). It may be also responsible for creation its own unique instance.

2.6 Design patterns in Control Automation

The existing design patterns for Control Automation in the literature are listed.

Dibowski et al. [21] propose a software design approach for large building automation systems (BAS). The design of large BASs with thousands of devices is a laborious task with a lot of recurrent works for identical automated rooms. The usage of prefabricated off-the-shelf devices and design patterns can simplify this task but creates new interoperability problems.

Serna et al. [22] present two specific design patterns which allow dealing with failure management in control applications based on IEC 61499 “extended function blocks”.

Dubinin et al. [23], suggest the use of design patterns that make applications robust to changes of execution semantics. A semantic-robust pattern is defined for a particular source execution model. The patterns are implemented by means of the same function block language apparat us and therefore are universal.

Serna et al. [24] presents a design pattern for machine tool control software applications. The design pattern suggests an indirect detection of deteriorations in the maintenance. In this manner adequate maintenance actions can be identified just in time. [28] describes first the design and operation phase followed by the maintenance phase.

Eckert and Fay et al. In [25] implemented an example of a liquid tank with use of design patterns in relation to the requirements and solution characteristics taking in consideration the functional and non-functional requirements.

Steinegger et al. [26] presented a design methodology to separate and decouple control code for normal operations from fault detection and fault handling methods in discrete manufacturing systems. Both resulting design patterns take hierarchical control software design into account and realize a centralized as well as a hierarchically-structured fault management. The proposed design was applied to an injection molding machine for evaluation demonstrating that design patterns help, beside enhancing the overall code quality and increasing maintainability, also to save time during the engineering process of industrial control applications.

Dai and Vyatkin in [27] proposed a component-based design pattern using an airport baggage handling system (BHS) as an example. The case of study shows the improvement in reusability when the proposed component-based design pattern is applied.

Racchetti et al. [28] Proposed a design pattern to direct map UML State-charts to IEC61131 code based on the previous work by [29]. This design pattern uses OO characteristics, providing an engine made of classes and mechanisms that allows developer to realize the static structure of State-charts.

Witsch and Vogel-Heuser in [1], state that pattern catalogues collect software design experience and up to now such design pattern catalogues for the domain of control engineering do not exist.

Something to notice about the literature is that, the first design patterns in software engineering [6], without considering the State design pattern used by Racchetti et al. [28], are not documented in the Automation Industry domain. These resources from IT technologies can be used to solve specific situations.

3 Procedure

3.1 Builder design pattern

In this chapter, two different implementations of the Builder design pattern will be described. The classical computer science approach to the Builder pattern and an optional approach to the Builder pattern that, on the author's opinion, could be more suited for industrial automation.

3.1.1 Classical approach to the Builder pattern

The classical approach to the Builder pattern assumes the ConcreteBuilder class is the only one in control of the blueprints for the final product; This way, the ConcreteBuilder class can build final products as long as there is a complete blueprint of the final product. Outside of these blueprints, no object can be created.

This approach can be used specially when there is a concrete amount of objects to be made and the client can not make any modifications to those objects when requesting them.

3.1.1.1 UML modeling

In this chapter, the classical approach to a simple Builder pattern will be described with UML notation.

To Start with, an activity diagram will show the scenario where an Builder pattern is needed.

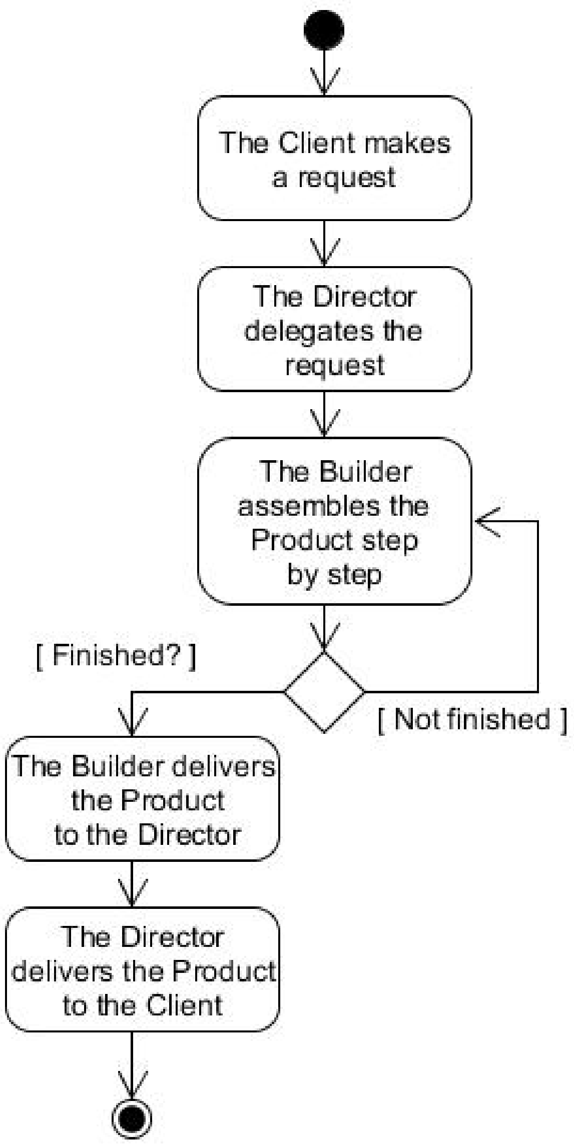

The Client makes a request
The Director delegates the request
The Builder assembles the Product step by step
[Finished?]
[Not finished]
The Builder delivers the Product to the Director
The Director delivers the Product to the Client

Figure 10: Builder pattern activity diagram

From the activity diagram, a state and sequence diagrams are proposed.

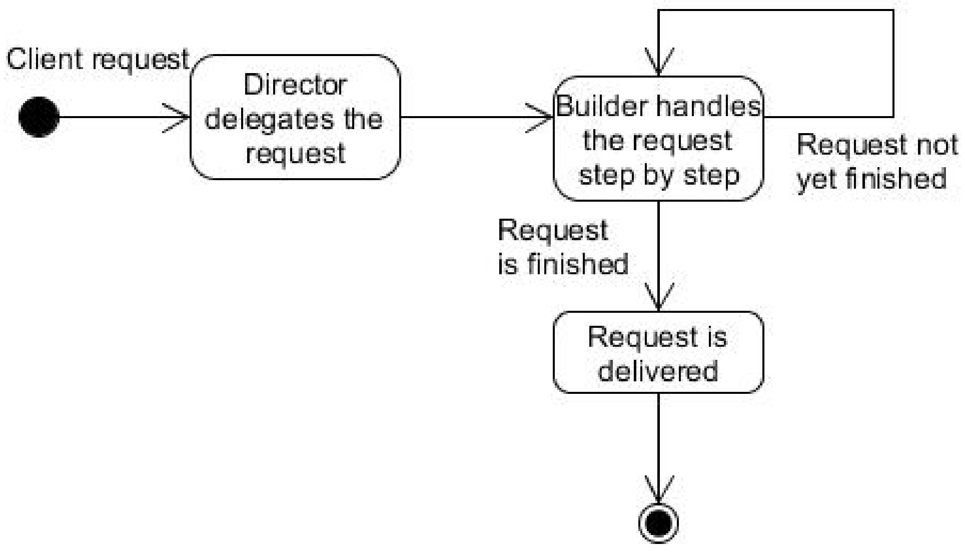


Figure 11: Builder pattern state diagram

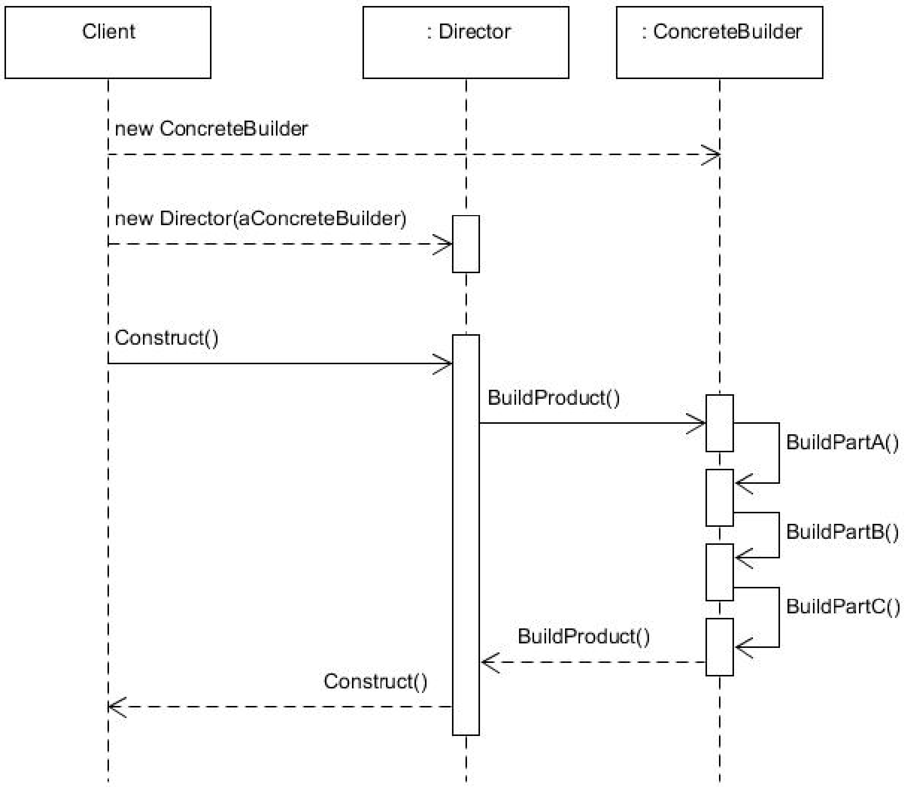


Figure 12: Builder pattern sequence diagram

As is noted on Figure : 12, the client sends the blueprint to the director in a request and the the client does the construct request expecting the final product. The director delegates the complete construction to the ConcreteBuilder class which, using the client's passed blueprint to the director, builds and delivers the final product to the director who then delivers it to the client.

Finally, the class diagram of a Builder pattern, to be implemented in the CODESYS IDE, is presented and its basic functionality is also described.

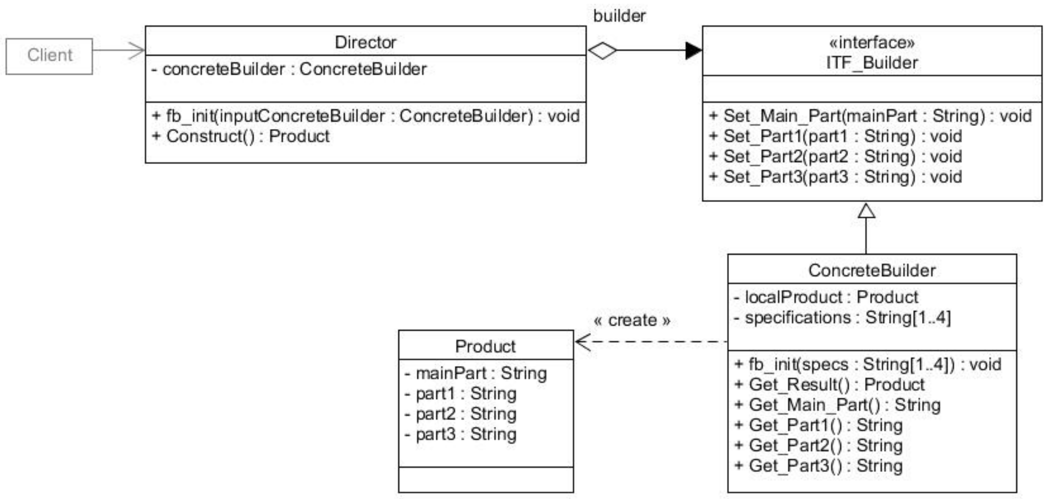

Client
builder
Director
- concreteBuilder : ConcreteBuilder
+ fb_init(inputConcreteBuilder : ConcreteBuilder) : void
+ Construct() : Product
«interface»
ITF_Builder
+ Set_Main_Part(mainPart : String) : void
+ Set_Part1(part1 : String) : void
+ Set_Part2(part2 : String) : void
+ Set_Part3(part3 : String) : void
ConcreteBuilder
- localProduct : Product
- specifications : String[1..4]
+ fb_init(specs : String[1..4]) : void
+ Get_Result() : Product
+ Get_Main_Part() : String
+ Get_Part1() : String
+ Get_Part2() : String
+ Get_Part3() : String
« create »
Product
- mainPart : String
- part1 : String
- part2 : String
- part3 : String

Figure 13: Builder template class diagram

- Director:
  - Variable *concreteBuilder : ConcreteBuilder*; This field will hold the correct ConcreteBuilder instance as a blueprint for the next object to be created. This variable is passed to the director directly from the client.
  - Method *fb_init (inputConcreteBuilder : ConcreteBuilder) : void*; The fb_init method is the default constructor of any Function Block in CODESYS. The only responsibility of the constructor in this class is to receive the blueprint in the *inputConcreteBuilder* variable and store it in its own *concreteBuilder* field. This constructor can be called by the same instance as many times as needed, this way a "Set" method for the concreteBuilder variable is not needed.
  - Method *Construct () : Product*; This method is used by the client to request the creation of the final product. As is noted in the syntax of the method, the Construct method must deliver an instance of the Product class.

- ITF_Builder:
  - Methods *Set_Main_Part (mainPart : String) : String*; This method will allow the write option to the *mainPart* field of the Product instance *localProduct* located within the ConcreteBuilder class.
  - Methods *Set_Part1 (part1 : String) : String*; This method will allow the write option to the *part1* field of the Product instance *localProduct* located within the ConcreteBuilder class.
  - Methods *Set_Part2 (part2 : String) : String*; This method will allow the write option to the *part2* field of the Product instance *localProduct* located within the ConcreteBuilder class.
  - Methods *Set_Part3 (part3 : String) : String*; This method will allow the write option to the *part3* field of the Product instance *localProduct* located within the ConcreteBuilder class.

- ConcreteBuilder:
  - Variable *localProduct : Product*; This variable is an instance of the Product POU. It is used by the ConcreteBuilder instance for the construction of the final product; Each individual step taken to the completion of the final product is first stored in this field.
  - Variable *specifications : String[1..4]*; This array holds the current blueprint to be used to deliver the final product instance.
  - Method *fb_init (specs : String[1..4]) : void*; The constructor of this class is used on the same way as the constructor of the director class. It changes the *specifications* field to the input variable *specs* passed by the client.
  - Method *Get_Result () : Product*; This method is used to deliver the *localProduct* object, when it is complete, as the final product object to the director instance and therefore, to the PLC_PRG or the client.
  - Method *Get_Main_Part () : String*; This method allows the read option of the variable *mainPart* in the *localProduct* object stored in the ConcreteBuilder instance.
  - Method *Get_Part1 () : String*; This method allows the read option of the variable *part1* in the *localProduct* object stored in the ConcreteBuilder instance.
  - Method *Get_Part2 () : String*; This method allows the read option of the variable *part2* in the *localProduct* object stored in the ConcreteBuilder instance.
  - Method *Get_Part3 () : String*; This method allows the read option of the variable *part3* in the *localProduct* object stored in the ConcreteBuilder instance.
- Product
  - Variable *mainPart : String*; This variable is used to store the main part value of a product instance.
  - Variable *part1 : String*; This variable is used to store the part1 value of a product instance.
  - Variable *part2 : String*; This variable is used to store the part2 value of a product instance.
  - Variable *part3 : String*; This variable is used to store the part3 value of a product instance.

Following this defined software architecture, a domain specific Decorator pattern code template is proposed.

3.1.1.2 Code template

The implementations was performed in the CODESYS IDE with the specific runtime: CODESYS Control for Raspberry Pi SL.

The code template for this pattern was developed following the class diagram described in the last section. It includes the Director POU, the ConcreteBuilder POU, the Product POU and the ITF_Builder interface as described in the UML class diagram.

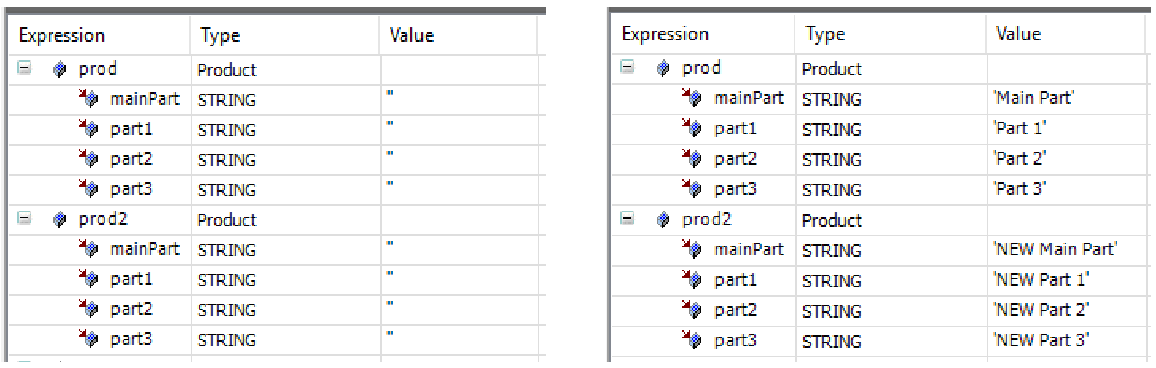
The code template PLC_PRG, the client, considers one director, two ConcreteBuilder and two product instances.

In the Figure : 14, one can see three request from the client. The requests on rungs 1 and 10, specify the concreteBuilder instances or "blueprint" to be used. On rung 7, the request for the final Product instances is demanded.

The director object makes the product object request to each ConcreteBuilder object without knowing the specifications of the final product object. Only the CocreteBuilder objects know its own default specifications and will deliver a product of its specifications every time a product request is made to them.

```
1       //The client sets the specifications to build a new Product object. This
        specifications are constant in concreteBuilder1
2       aDirector ( inputConcreteBuilder  := concreteBuilder1 ) ;
3
4       //The client asks the Director for the final Product object after he has set the
         builder specifications.
5       //The Director delegates the construction of the Product object to the Builder
        object and delivers the Product
6       // object after the Builder has finished "building" the Product object
7       prod := aDirector . Construct ( ) ;
8
9       //The client changes the specifications to build a new Product object
10      aDirector ( inputConcreteBuilder  := concreteBuilder2 ) ;
```

Figure 14: Client's request code



Expression	Type	Value
prod	Product	
mainPart	STRING	''
part1	STRING	''
part2	STRING	''
part3	STRING	''
prod2	Product	
mainPart	STRING	''
part1	STRING	''
part2	STRING	''
part3	STRING	''

(a) Product objects before one RUN cycle

Expression	Type	Value
prod	Product	
mainPart	STRING	'Main Part'
part1	STRING	'Part 1'
part2	STRING	'Part 2'
part3	STRING	'Part 3'
prod2	Product	
mainPart	STRING	'NEW Main Part'
part1	STRING	'NEW Part 1'
part2	STRING	'NEW Part 2'
part3	STRING	'NEW Part 3'

(b) Product objects after one RUN cycle

Figure 15: Classic Builder code template sample results

As is noted in figure: 15, after one RUN cycle the product objects are created with the specifications of the ConcreteBuilder object they were built on.

The Table 1 displays the overall results of the application after 700000 cycles. For the testing environment we are comparing the different results between a simulation environment in CODESYS and the real hardware running on a Raspberry Pi 3 model B+ with the CODESYS Control for Raspberry Pi SL runtime.

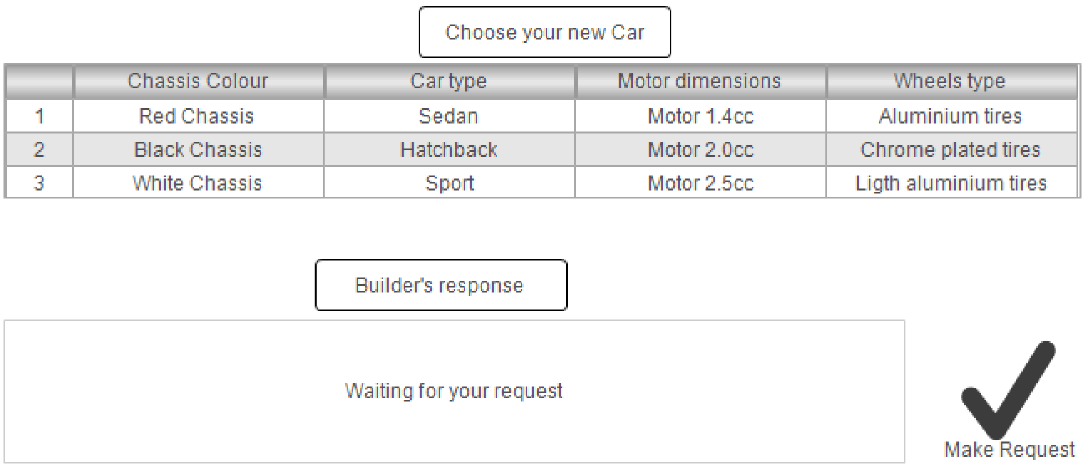
Table 1: Overall results of the classical Builder code template application

Description	Simulation	Raspberry Pi
Allocated memory size	90KB	130KB
Base cycle time	10ms	10ms
Average cycle time	20us	179us
Maximum cycle time	738us	418us
Maximum Jitter	14248us	229us

The full code project can be found in section 6.1 Appendix A given in the Documentation format directly from the CODESYS environment.

3.1.1.3 Industrial application

As an industrial application, a Master HMI with three final product possibilities to chose from is proposed.



Choose your new Car

	Chassis Colour	Car type	Motor dimensions	Wheels type
1	Red Chassis	Sedan	Motor 1.4cc	Aluminium tires
2	Black Chassis	Hatchback	Motor 2.0cc	Chrome plated tires
3	White Chassis	Sport	Motor 2.5cc	Ligth aluminium tires

Builder's response

Waiting for your request

Figure 16: Builder application HMI

This HMI, Figure: 16, template replaces the PLC_PRG as the client. The client can make requests by choosing one option of the HMI and then clicking the request button. Once a request has been made, the PLC_PRG will handle the specific request and will build the desired product object. The result of each request will be displayed on text boxes below.

To complete the implementation, a few changes are considered in the class diagram Figure:17.

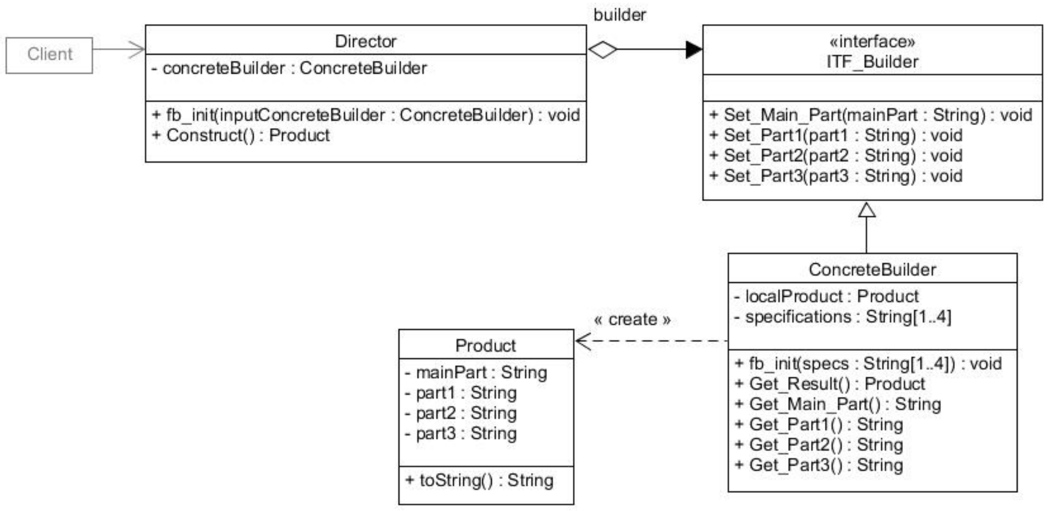

Client
Director
- concreteBuilder : ConcreteBuilder
+ fb_init(inputConcreteBuilder : ConcreteBuilder) : void
+ Construct() : Product
builder
«interface»
ITF_Builder
+ Set_Main_Part(mainPart : String) : void
+ Set_Part1(part1 : String) : void
+ Set_Part2(part2 : String) : void
+ Set_Part3(part3 : String) : void
ConcreteBuilder
- localProduct : Product
- specifications : String[1..4]
+ fb_init(specs : String[1..4]) : void
+ Get_Result() : Product
+ Get_Main_Part() : String
+ Get_Part1() : String
+ Get_Part2() : String
+ Get_Part3() : String
« create »
Product
- mainPart : String
- part1 : String
- part2 : String
- part3 : String
+ toString() : String

Figure 17: Builder application class diagram

- Product
  - Method *toString () : String*; Method that displays the parameters of the calling instance in one string variable in order to be displayed by the HMI.

The Table 2 displays the overall results of the application with the same specifications of the last test. As most of the time the application will be on idle, the average cycle time in the simulation environment can be neglected.

Table 2: Overall results of the classical Builder industrial application

Description	Simulation	Raspberry Pi
Allocated memory size	1544KB	2235KB
Base cycle time	10ms	10ms
Average cycle time	n/a	43us
Maximum cycle time	914us	223us
Maximum Jitter	11203us	148us

The full code project can be found in section 6.2 Appendix B given in the Documentation format directly from the CODESYS environment.

3.1.2 Optional approach to the Builder pattern

The optional approach to the Builder pattern retrieves the control of the blueprints from the ConcreteBuilder class and grants it to the director class. This way, the director class has full control and instead of making blueprints of the object, it can deliver a fully personalized product instance.

This approach can be used specially when there is an unlimited[1] amount of objects to be made and the client can make any modifications or special demands to those objects when requesting them.

3.1.2.1 UML modeling

In this chapter, an optional approach to a simple Builder pattern will be described with UML notation.

To Start with, an activity diagram will show the scenario where an Builder pattern is needed.

[1]Limited only by the amount of "parts" possible to be assembled.

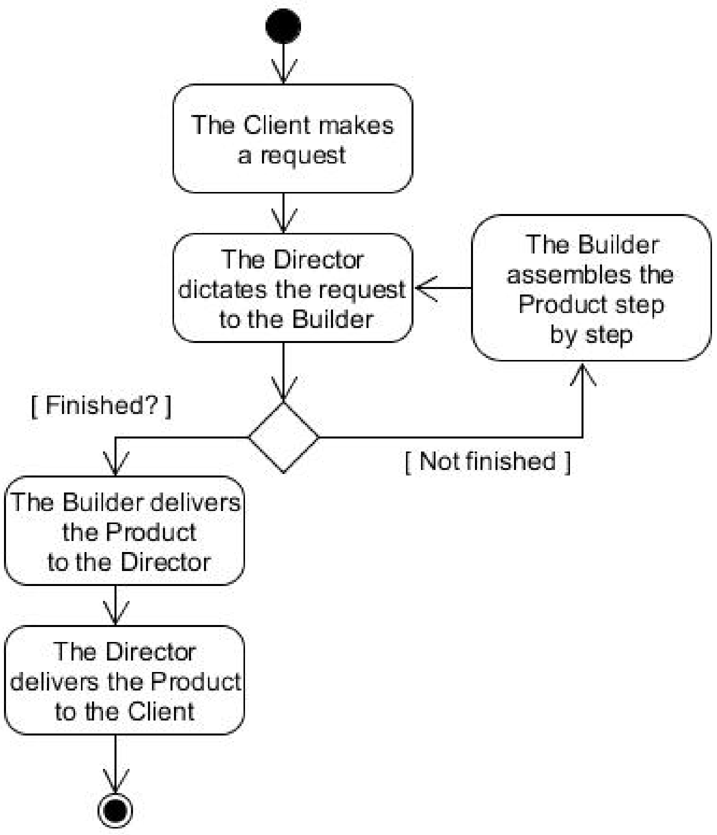

The Client makes a request
The Director dictates the request to the Builder
The Builder assembles the Product step by step
[Finished?]
[Not finished]
The Builder delivers the Product to the Director
The Director delivers the Product to the Client

Figure 18: Builder optional activity diagram

From the activity diagram, a state and sequence diagrams are proposed.

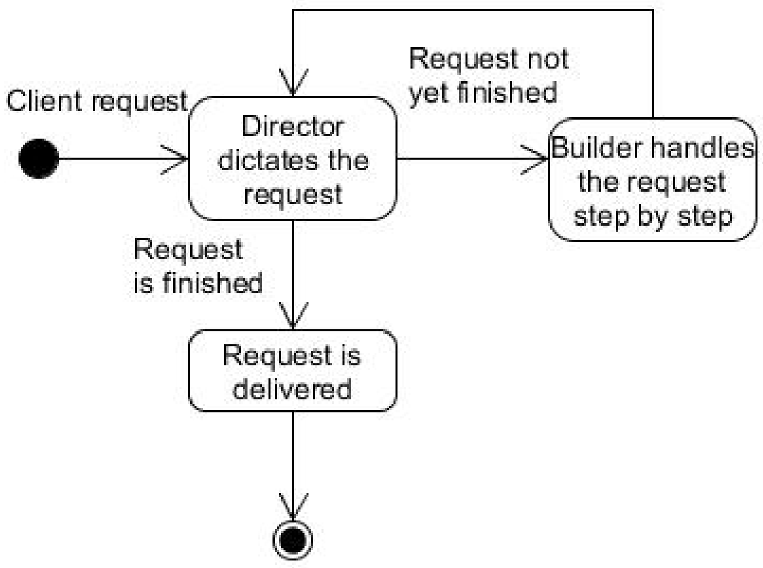


Figure 19: Builder optional state diagram

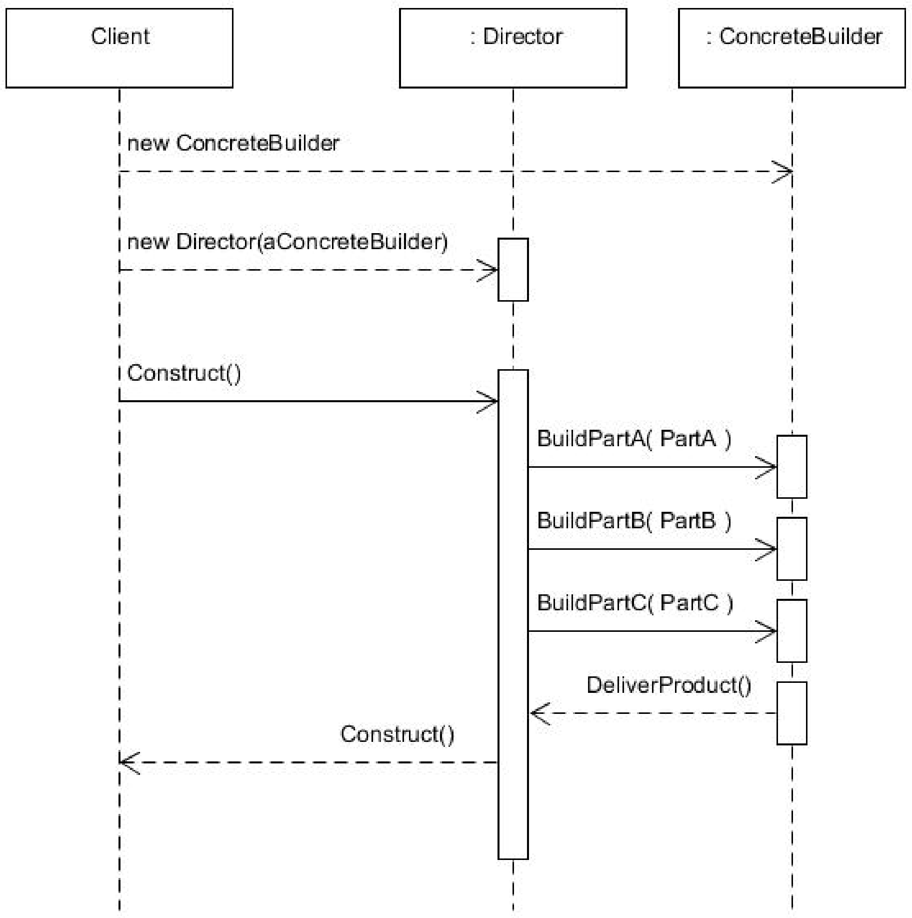


Figure 20: Builder optional sequence diagram

As is noted on Figure : 20, the client sends a ConcreteBuilder instance to the director but in this case, this ConcreteBuilder instance will work just as the "workshop" to build the final product object. The director, delegates the constructions of the final product to its ConcreteBuider instances but it is the director instance who "pushes" the concrete specifications from the client to the ConcreteBuilder instance. At the end, the Concrete-Builder instances builds the final product objects and delivers it to the director just to hand it over to the client.

Finally, the class diagram of a Builder pattern, to be implemented in the CODESYS IDE, is presented and its basic functionality is also described[2].

[2]Only the differences to the classical approach will be described.

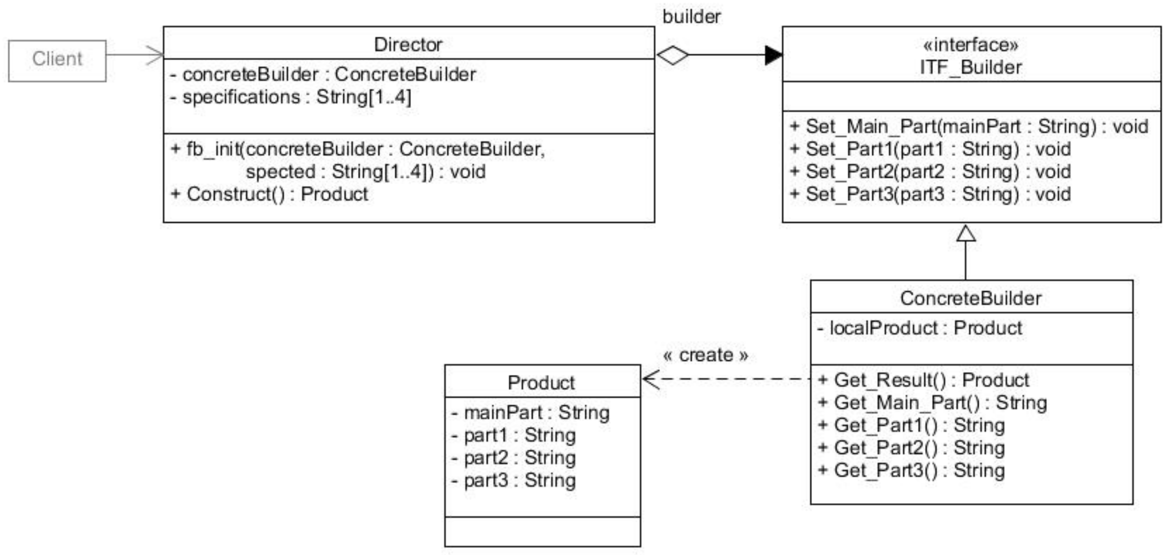

Client
Director
- concreteBuilder : ConcreteBuilder
- specifications : String[1..4]
+ fb_init(concreteBuilder : ConcreteBuilder, spected : String[1..4]) : void
+ Construct() : Product
builder
«interface»
ITF_Builder
+ Set_Main_Part(mainPart : String) : void
+ Set_Part1(part1 : String) : void
+ Set_Part2(part2 : String) : void
+ Set_Part3(part3 : String) : void
ConcreteBuilder
- localProduct : Product
+ Get_Result() : Product
+ Get_Main_Part() : String
+ Get_Part1() : String
+ Get_Part2() : String
+ Get_Part3() : String
« create »
Product
- mainPart : String
- part1 : String
- part2 : String
- part3 : String

Figure 21: Builder optional template class diagram

- Director
  - Variable *specifications : String[1..4]*; The *specifications* variable, that on the ConcreteBuilder had on the classical approach, has been transferred to the director class. It holds the concrete specifications from the client for the constructions of the final product object.
  - Method *fb_init (concreteBuilder : ConcreteBuilder , spected : String[1..4]) : void*; The only difference in the constructor method is that now it receives, as the *spected* input variable, the full blueprint from the client.

3.1.2.2 Code template

The implementations was performed in the CODESYS IDE with the specific runtime: CODESYS Control for Raspberry Pi SL.

The code template for this pattern was developed following the class diagram described in the last section. It includes the Director POU, the ConcreteBuilder POU, the Product POU and the ITF_Builder interface as described in the UML class diagram.

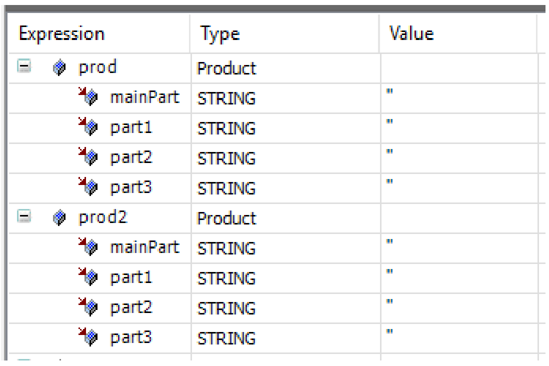
The code template PLC_PRG, the client, considers one director, one ConcreteBuilder and two product instances.

In the Figure : 22, one can see three request from the client. The requests on rungs 4 and 12, are the request to the construction of the final product objects, on this template, the specifications for the first object were preloaded. On rung 7, the client specifies to the aDirector instance the "blueprint" to be used next.

The director object makes the product object request to the ConcreteBuilder object passing it the specifications of the final product object. In this case, the director knows the specifications for every product object before it is built. The ConcreteBuilder object receives the specifications, builds the product object and delivers it to the director object. As the director dictates the specifications for the product to be built, there is only need to one ConcreteBuilder object.

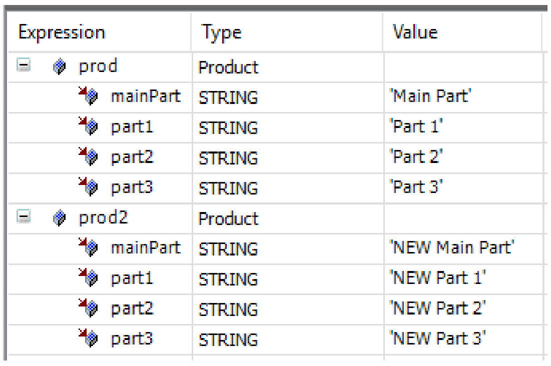
```
//The client asks the Director for the final Product object with the default specifications and builder.
//The Director delegates the construction of the Product object to the Builder object and delivers the Product
// object after the Builder has finished "building" the Product object
prod := aDirector . Construct ( ) ;

//The client changes the specifications to build a new Product object
aDirector ( spected := specs2 ) ;

//The client asks the Director for the final Product object with the new specifications.
//Again, the Director delegates the construction of the Product object to the Builder object and delivers the Product
// object after the Builder has finished "building" the Product object
prod2 := aDirector . Construct ( ) ;
```

Figure 22: Client's request code



Expression	Type	Value
prod	Product	
mainPart	STRING	''
part1	STRING	''
part2	STRING	''
part3	STRING	''
prod2	Product	
mainPart	STRING	''
part1	STRING	''
part2	STRING	''
part3	STRING	''

(a) Product objects before one RUN cycle



Expression	Type	Value
prod	Product	
mainPart	STRING	'Main Part'
part1	STRING	'Part 1'
part2	STRING	'Part 2'
part3	STRING	'Part 3'
prod2	Product	
mainPart	STRING	'NEW Main Part'
part1	STRING	'NEW Part 1'
part2	STRING	'NEW Part 2'
part3	STRING	'NEW Part 3'

(b) Product objects after one RUN cycle

Figure 23: Optional Builder code template sample results

As is noted in figure: 23, after one RUN cycle the product objects are created with the specifications given to the ConcreteBuilder object by the director object.

The Table 3 displays the overall results of the application after 700000 cycles. For the testing environment we are comparing the different results between a simulation environment in CODESYS and the real hardware running on a Raspberry Pi 3 model B+ with the CODESYS Control for Raspberry Pi SL runtime.

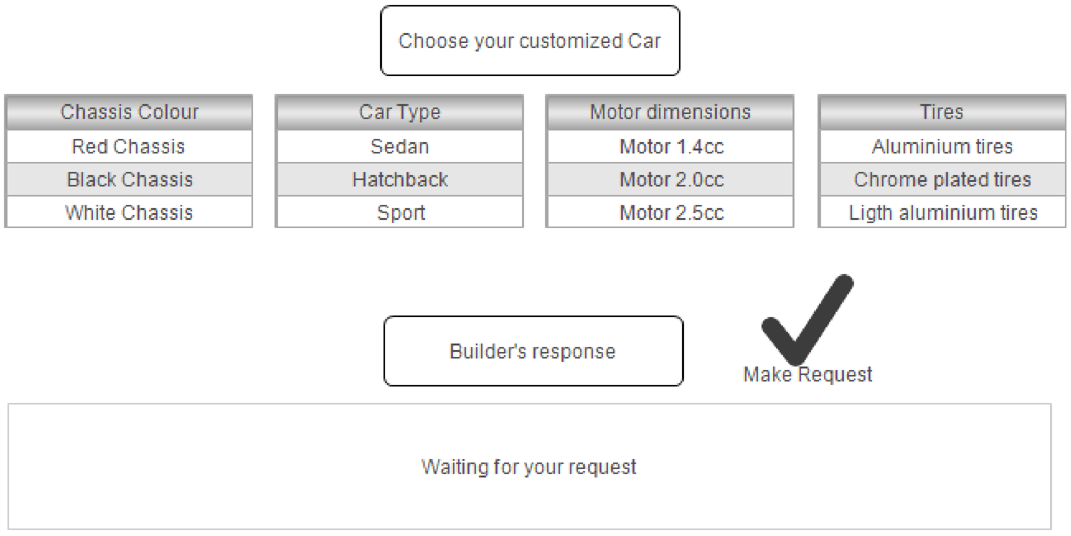
Table 3: Overall results of the optional Builder code template application

Description	Simulation	Raspberry Pi
Allocated memory size	88KB	128KB
Base cycle time	10ms	10ms
Average cycle time	19us	160us
Maximum cycle time	1103us	507us
Maximum Jitter	13979us	191us

The full code project can be found in section 6.3 Appendix C given in the Documentation format directly from the CODESYS environment.

3.1.2.3 Industrial application

As an industrial application, a Master HMI with four tables to fully customize a final product is proposed.



Choose your customized Car

Chassis Colour	Car Type	Motor dimensions	Tires
Red Chassis	Sedan	Motor 1.4cc	Aluminium tires
Black Chassis	Hatchback	Motor 2.0cc	Chrome plated tires
White Chassis	Sport	Motor 2.5cc	Ligth aluminium tires

Builder's response

Make Request

Waiting for your request

Figure 24: Builder optional application HMI

This HMI template, Figure: 24, replaces the PLC_PRG as the client. The client can make requests by choosing one option on every table of the HMI and then clicking the request button. Once a request has been made, the PLC_PRG will handle the specific request and will build the desired product object. The result of each request will be displayed on text boxes below.

To complete the implementation, a few changes are considered in the class diagram Figure:25.

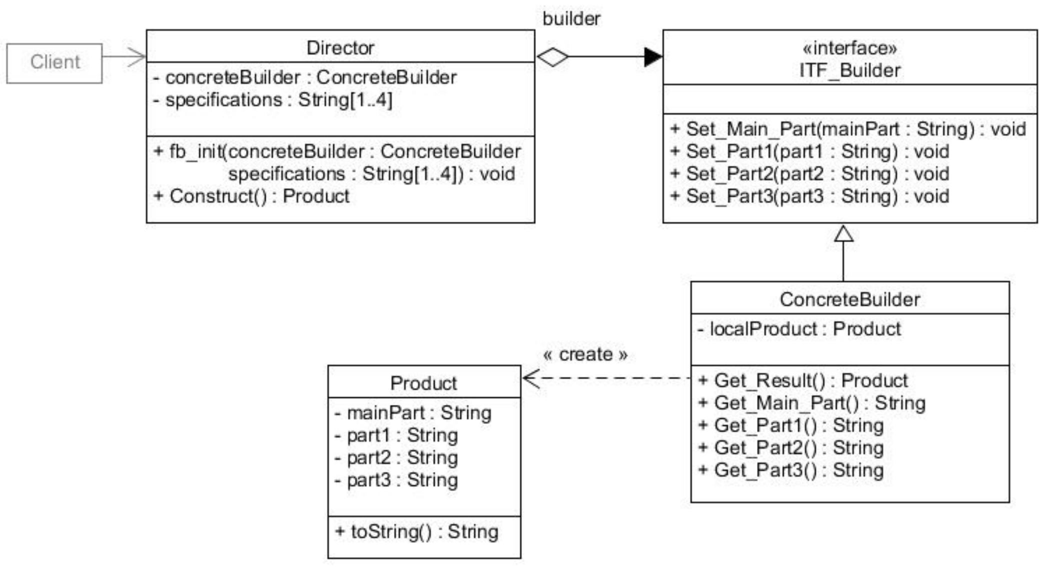

Client
builder
Director
- concreteBuilder : ConcreteBuilder
- specifications : String[1..4]
+ fb_init(concreteBuilder : ConcreteBuilder specifications : String[1..4]) : void
+ Construct() : Product
«interface» ITF_Builder
+ Set_Main_Part(mainPart : String) : void
+ Set_Part1(part1 : String) : void
+ Set_Part2(part2 : String) : void
+ Set_Part3(part3 : String) : void
ConcreteBuilder
- localProduct : Product
+ Get_Result() : Product
+ Get_Main_Part() : String
+ Get_Part1() : String
+ Get_Part2() : String
+ Get_Part3() : String
« create »
Product
- mainPart : String
- part1 : String
- part2 : String
- part3 : String
+ toString() : String

Figure 25: Builder optional application class diagram

- Product

- Method *toString () : String*; This method displays the parameters of the calling instance in one string variable in order to be displayed by the HMI.

The Table 4 displays the overall results of the application with the same specifications of the last test. As most of the time the application will be on idle, the average cycle time in the simulation environment can be neglected.

Table 4: Overall results of the optional Builder industrial application

Description	Simulation	Raspberry Pi
Allocated memory size	1562KB	2254KB
Base cycle time	10ms	10ms
Average cycle time	n/a	46us
Maximum cycle time	957us	192us
Maximum Jitter	8669us	131us

The full code project can be found in section 6.4 Appendix D given in the Documentation format directly from the CODESYS environment.

3.2 Decorator design pattern

Since the industrial revolution, ordinary machines have been designed for mass production. Nowadays, flexible machines are able to manufacture a certain amount of products but, most of the times, one at the time until the demanded production is reached. One of the core ideas of the Industry 4.0, on the marketing side, is the total control and personification of the desired product. The Decorator pattern can be used to provide added functionality to machines at run-time. This means, the machines could be able to apply the specific decorators to the product without altering the other products in line, making each product unique.

3.2.1 UML modeling

In this chapter, a simple Decorator pattern will be described with UML notation.

To Start with, an activity diagram will show the scenario where an Decorator pattern is needed.

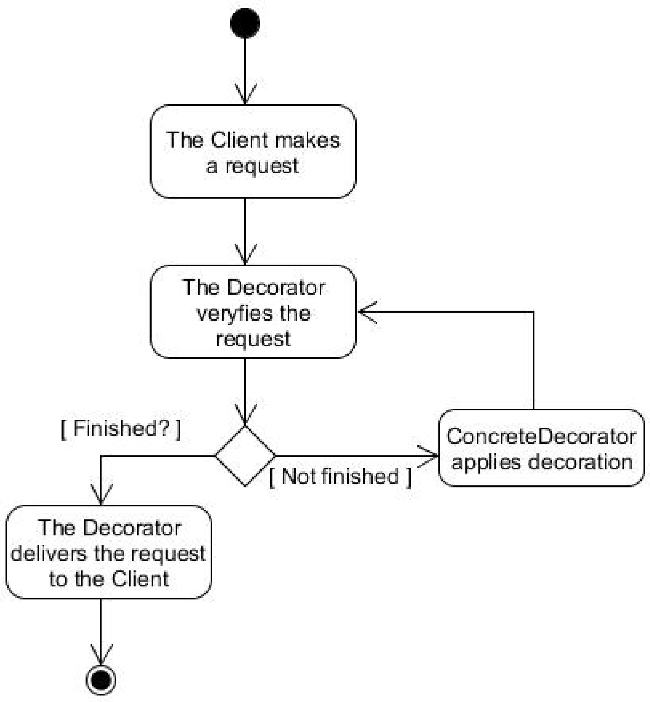

The Client makes a request
The Decorator veryfies the request
[Finished?]
[Not finished]
ConcreteDecorator applies decoration
The Decorator delivers the request to the Client

Figure 26: Decorator activity diagram

From the activity diagram, a state and sequence diagrams are proposed.

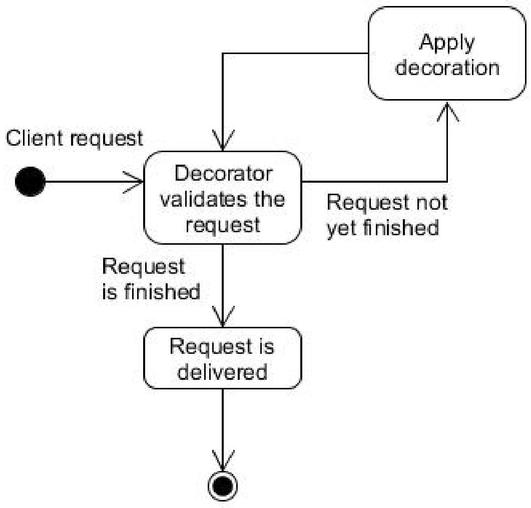


Figure 27: Decorator state diagram

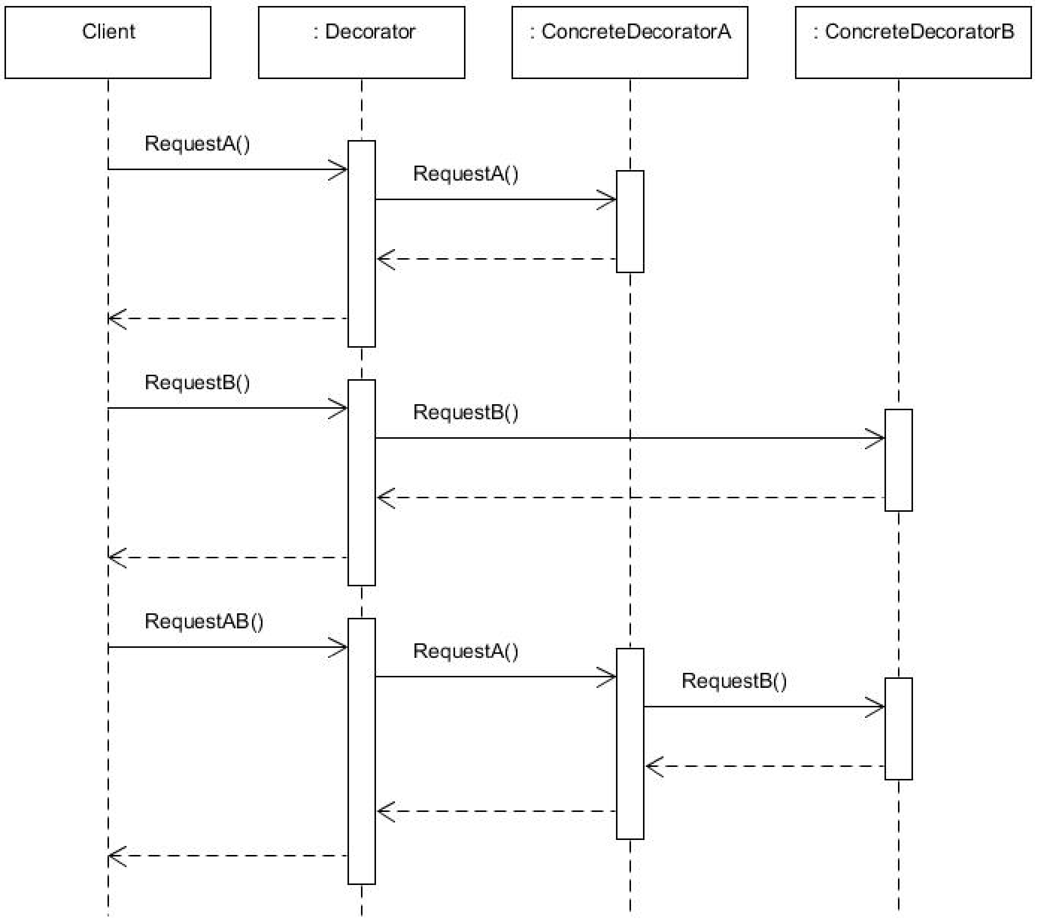


Figure 28: Decorator sequence diagram

As is noted on Figure : 28, the client can send an specific request, either A, B or AB; then, the decorator class and its dependants must handle these requests and apply the concrete decorators to the final product.

Finally, the class diagram of a Decorator pattern, to be implemented in the CODESYS IDE, is presented and its basic functionality is also described.

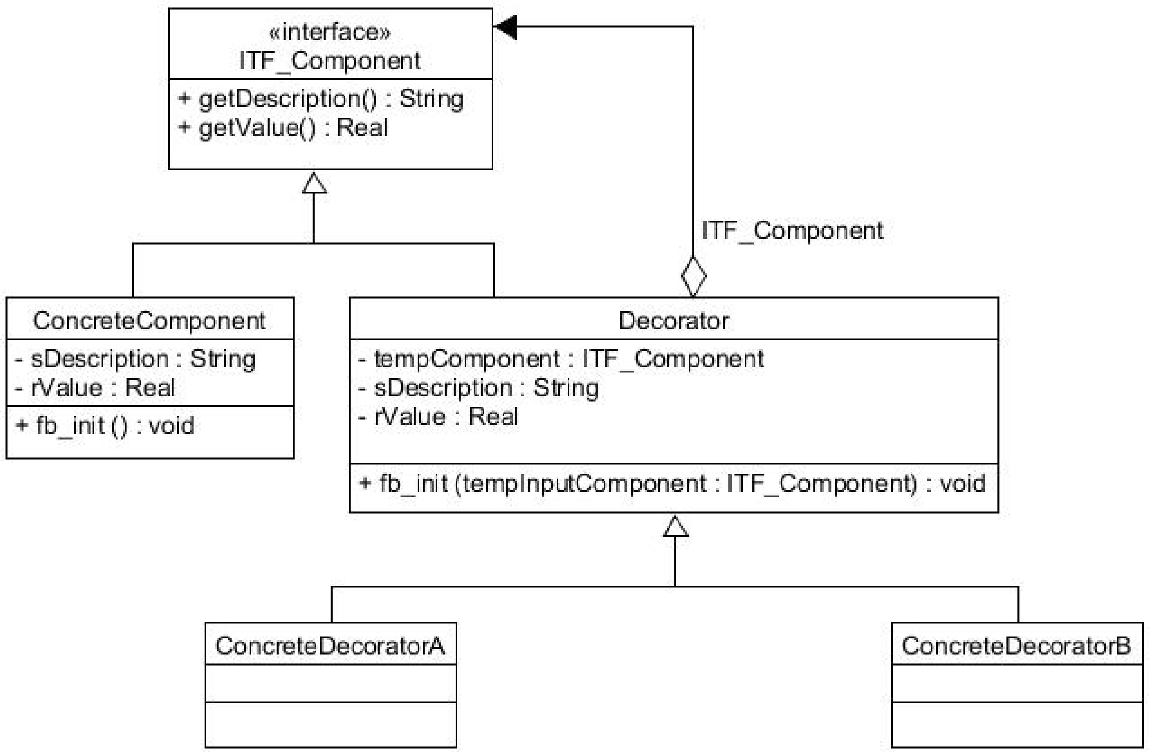


Figure 29: Decorator template class diagram

- ITF Component:
  - Method *getDescription () : String*; This method will allow the read option of the *sDescription* field on the classes ConcreteComponent and decorator. On the case of the ConcreteDecorator classes, this method will be responsible of adding functionality at runtime.
  - Method *getValue () : Real*; This method will allow the read option of the *rValue* field on the classes ConcreteComponent and decorator. On the case of the ConcreteDecorator classes, this method will be responsible of adding functionality at runtime.
- ConcreteComponent:
  - Variable *sDescription : String*; This field will be decorated in accordance to the request from the client.
  - Variable *rValue : Real*; This field will be decorated in accordance to the request from the client.
  - Method *fb_init () : void*; The constructor of the ConcreteComponent POU handles the correct initialization of the variables *sDescription* and *rValue* with its default values.
- Decorator:
  - Variable *tempComponent : ITF_Component*; This variable holds a reference to the object being decorated. This can be an already decorated object or a plain component.

- Variable *sDescription : String*; This field will be decorated in accordance to the request from the client.
- Variable *rValue : Real*; This field will be decorated in accordance to the request from the client.
- Method *fb_init (tempInputComponent : ITF_Component) : void*; The constructor of the decorator and ConcreteDecorator classes does two things. It saves the object to be decorated, passed as the *tempInputComponent* variable, to its own field *tempComponent*; and it handles the correct initialization of the variables *sDescription* and *rValue* with its default values for each ConcreteDecorator class.

Following this defined software architecture, a domain specific Decorator pattern code template is proposed.

3.2.2 Code template

The implementations was performed in the CODESYS IDE with the specific runtime: CODESYS Control for Raspberry Pi SL.

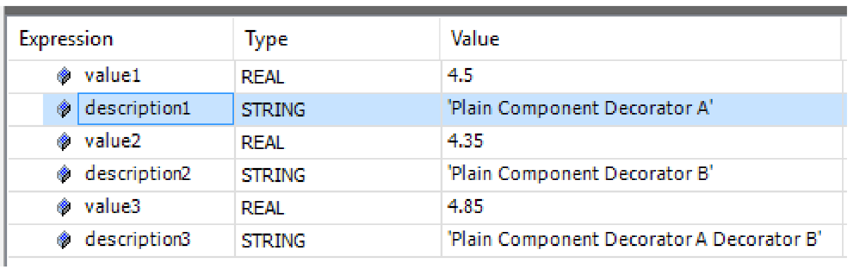
The code template for this pattern was developed following the class diagram described in the last section. It includes the ConcreteComponent POU, the Decorator POU, the ITF_Component interface and two ConcreteComponent POUs as described in the UML class diagram.

The code template PLC_PRG, the client, considers one ConcreteComponent object and three ConcreteDecorator instances.

In the code template, the default values for a ConcreteComponent object are: Value 4.0 and Description "Plain Component". The PLC_PRG has the necessary instructions, see Figure: 30, to display the values of the ConcreteComponent object after each decoration has been made as shown in the following figure 31.

```
//Results from the ConcreteDecoratorA class decoration
value1 := decorA . getValue ( ) ;
description1  := decorA . getDescription ( ) ;

//Results from the ConcreteDecoratorB class decoration
value2 := decorB . getValue ( ) ;
description2  := decorB . getDescription ( ) ;

//Results from both decorations
value3 := decorB2 . getValue ( ) ;
description3  := decorB2 . getDescription ( ) ;

```

Figure 30: Client's request code



Expression	Type	Value
value1	REAL	4.5
description1	STRING	'Plain Component Decorator A'
value2	REAL	4.35
description2	STRING	'Plain Component Decorator B'
value3	REAL	4.85
description3	STRING	'Plain Component Decorator A Decorator B'

Figure 31: Decorator template sample results

The figure: 31 shows the results of using the ConcreteDecorator classes with either a plain component or a decorated one[3]. The REAL values of the *rValue* fields on the ConcreteDecorator classes are 0.5 and 0.35 respectively; The STRING values of the *sDescription* fields are "Decorator A" and "Decorator B" respectively.
The Table 5 displays the overall results of the application after 700000 cycles. For the testing environment we are comparing the different results between a simulation environment in CODESYS and the real hardware running on a Raspberry Pi 3 model B+ with the CODESYS Control for Raspberry Pi SL runtime.

Table 5: Overall results of the Decorator code template application

Description	Simulation	Raspberry Pi
Allocated memory size	87KB	126KB
Base cycle time	10ms	10ms
Average cycle time	20us	139us
Maximum cycle time	699us	389us
Maximum Jitter	13020us	209us

The full code project can be found in section 6.5 Appendix E given in the Documentation format directly from the CODESYS environment.

3.2.3 Industrial application

As an industrial application, a Master HMI in charge of decorating the subject's value (INT) and description (String) is proposed.

[3]decorB2 takes a ConcreteDecoratorA instance as an input at instanciation

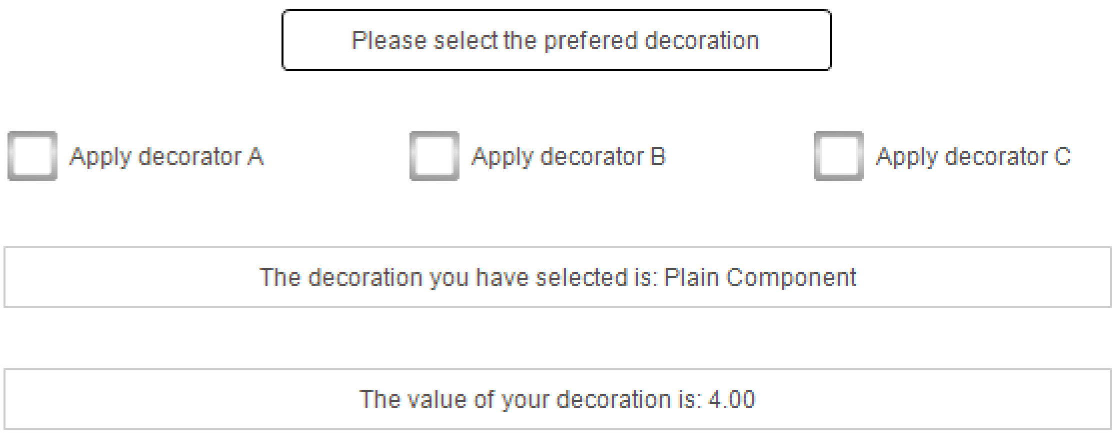

Please select the prefered decoration
Apply decorator A
Apply decorator B
Apply decorator C
The decoration you have selected is: Plain Component
The value of your decoration is: 4.00

Figure 32: Decorator application HMI

This HMI, Figure: 32, template replaces the PLC_PRG as the client. It can make requests by checking or clearing the checkboxes of the HMI. As soon as the previous state of a checkbox is changed, a request is made to the PLC_PRG.

The PLC_PRG will handle those requests on real-time by dynamically creating the requested object with its specific decorations. The result of each request will be displayed on text boxes below.

To complete the implementation, a few changes are considered in the class diagram Figure:33.

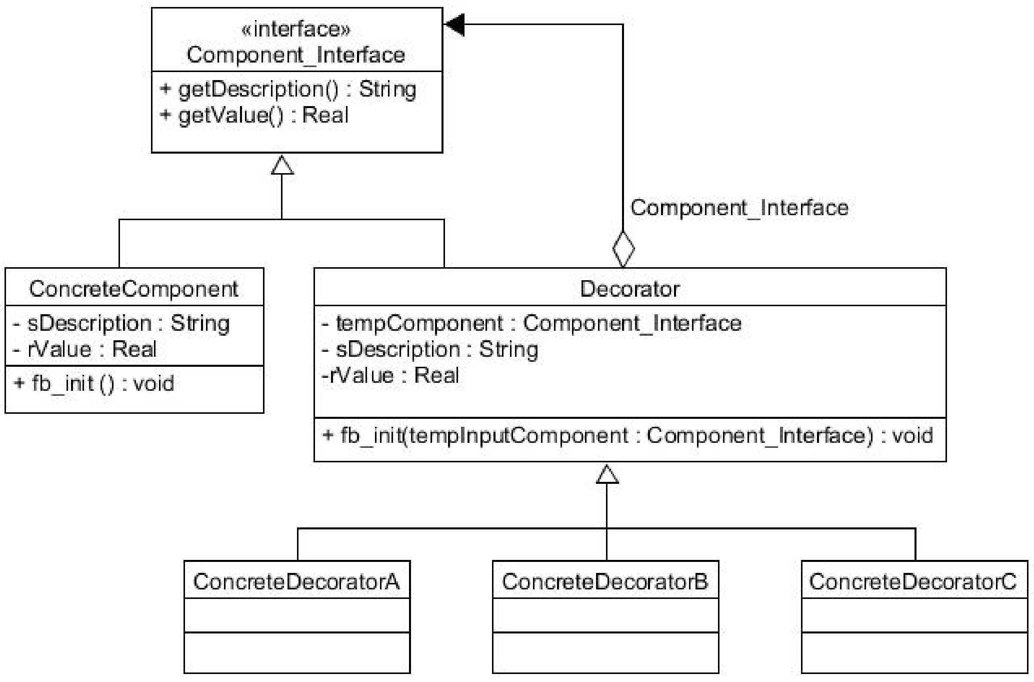


Figure 33: Decorator application class diagram

- ConcreteDecoratorC; Another ConcreteDecorator POU is implemented with the same properties as the other ConcreteDecorator's.

The Table 6 displays the overall results of the application with the same specifications of the last test. As most of the time the application will be on idle, the average cycle time in the simulation environment can be neglected.

Table 6: Overall results of the Decorator industrial application

Description	Simulation	Raspberry Pi
Allocated memory size	1289KB	1834KB
Base cycle time	10ms	10ms
Average cycle time	n/a	74us
Maximum cycle time	840us	271us
Maximum Jitter	4844us	135us

The full code project can be found in section 6.6 Appendix F given in the Documentation format directly from the CODESYS environment.

3.3 Observer design pattern

Due to the tendencies of Industry 4.0 and the Internet of Things, the communication between devices is crucial.

The Observer design pattern focuses on the automatic communication between devices or stations. The real difference between implementing a Observer design pattern and a broadcast communication in the network is, that by using the Observer pattern, the communication network will not be floated with messages to devices that do not require to be notified as in a broadcast message.

3.3.1 UML modeling

In this chapter, a simple Observer pattern will be described with UML notation.

To Start with, an activity diagram will show the scenario where an Observer pattern is needed.

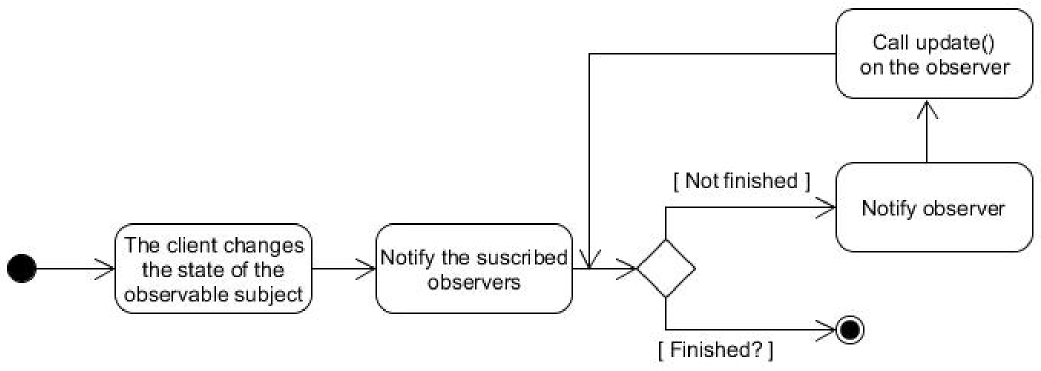

The client changes the state of the observable subject
Notify the suscribed observers
[Not finished]
Notify observer
Call update() on the observer
[Finished?]

Figure 34: Observer activity diagram

From the activity diagram, a state and sequence diagrams are proposed.

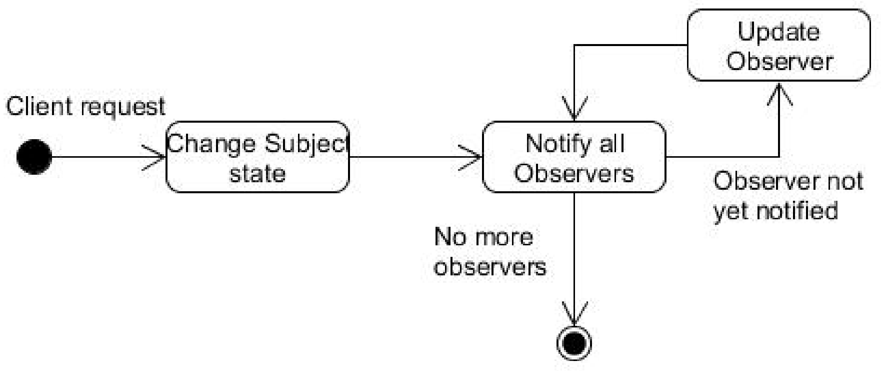


Figure 35: Observer state diagram

The proposed sequence diagram implements a push type notification. Where the subject object pushes the changes to the observers (writes to the observers) rather than the observers pulling the state change from the subject (reading from the subject).

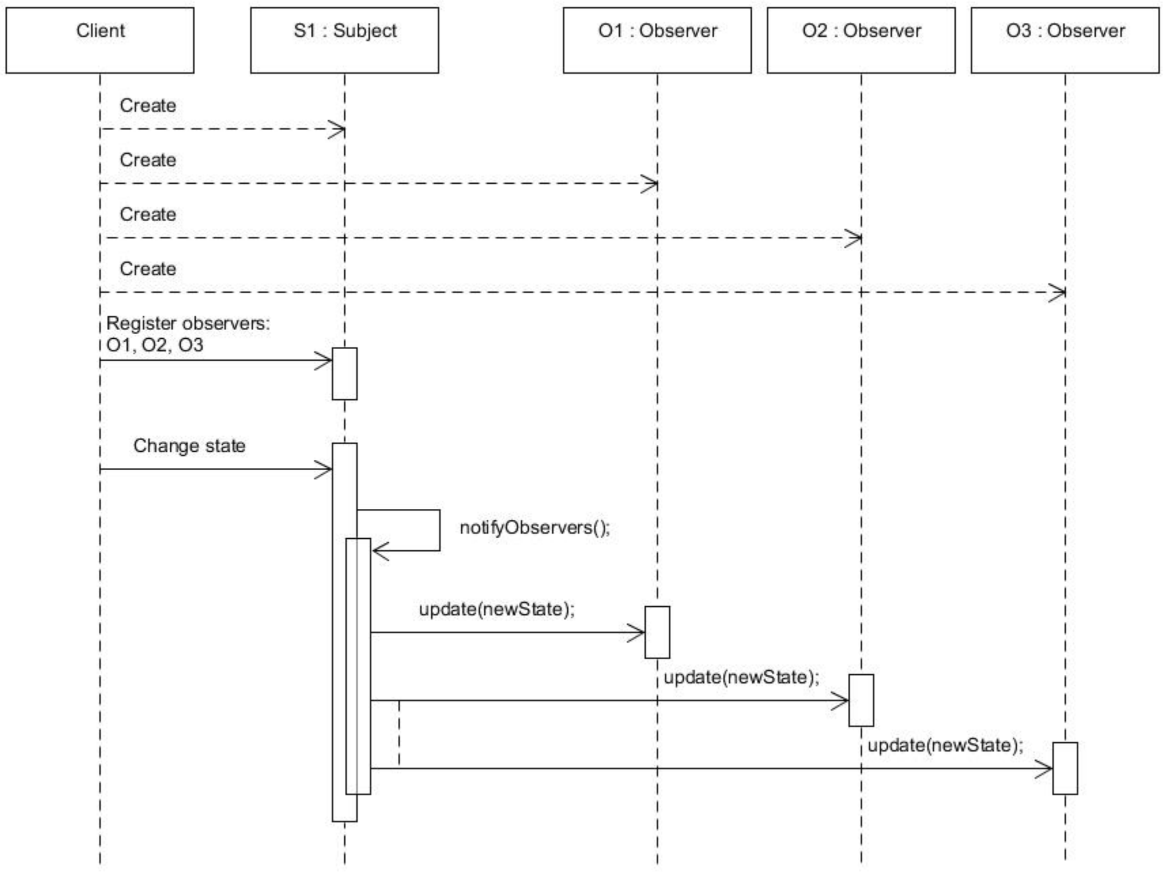


Figure 36: Observer sequence diagram

As is noted on Figure : 36, the subject instance is responsible of the notification of all its subscribed observers without taking into account who these observers are. As long as an observer is subscribed, it should be notified by the subject.

Finally, the class diagram of an Observer pattern, to be implemented in the CODESYS IDE, is presented and its basic functionality is also described.

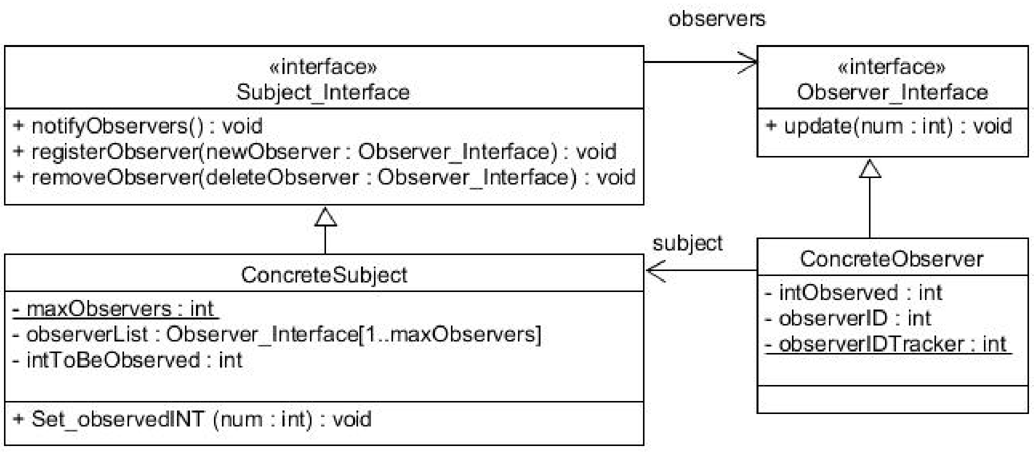

observers
«interface»
Subject_Interface
+ notifyObservers() : void
+ registerObserver(newObserver : Observer_Interface) : void
+ removeObserver(deleteObserver : Observer_Interface) : void
«interface»
Observer_Interface
+ update(num : int) : void
subject
ConcreteSubject
- maxObservers : int
- observerList : Observer_Interface[1..maxObservers]
- intToBeObserved : int
+ Set_observedINT (num : int) : void
ConcreteObserver
- intObserved : int
- observerID : int
- observerIDTracker : int

Figure 37: Observer template class diagram

- ConcreteSubject:
  - Constant static variable *maxObservers : int*; This variable is the maximum number of subscribed observers. This is due to the lack of arrays with variable length on CODESYS.
  - Variable *observerList : Observer_Interface[1..maxObservers]*; This array holds a list of all the subscribed observers to be used by the *notifyObservers()* method for the automatic notification of the observers.
  - Variable *intToBeObserved : int*; This is just a state variable; It will trigger the automatic notification every time it is subject to change.
  - Method *notifyObservers : int*; This method holds the logic behind the automatic notification to the subscribed observers. It uses a FOR loop and the *observerList* array to do so.
  - Method *registerObserver (newObserver : Observer_Interface) : void*; Method used to subscribe ConcreteObserver instances to the *observerList* array. The *newObserver* variable is the passed ConcreteObserver instance to be saved in the array.
  - Method *removeObserver (deleteObserver : Observer_Interface) : void*; Method used to unsubscribe ConcreteObserver instances to the *observerList* array. The *deleteObserver* variable is the passed ConcreteObserver instance to be removed from the array.
  - Method *Set_observedINT (num : int) : void*; This method is used to change the variable *intToBeObserved* of a ConcreteSubject instance

- ConcreteObserver:
  - Variable *intObserved : int*; Integer value to receive the automatic state notification.
  - Variable *observerID : int*; This variable works as the unique identifier of the ConcreteObserver instance. It is granted at the creation of the instance whether the instance is subscribed or not to an ConcreteSubject instance. This variable is not an essential part of the pattern but it is used to exemplify good code practices.

- Static variable *observerIDTracker : int*; This static variable total amount of ConcreteObserver instances in the system. This variable is not an essential part of the pattern but it is used to exemplify good code practices.
- Method *update (num : int) : void*; Method used by the ConcreteObserver instance to receive the changes on the observed variable from the ConcreteSubject instance.

Following this defined software architecture, a domain specific Observer pattern code template is proposed.

3.3.2 Code template

The implementations was performed in the CODESYS IDE with the specific runtime: CODESYS Control for Raspberry Pi SL.

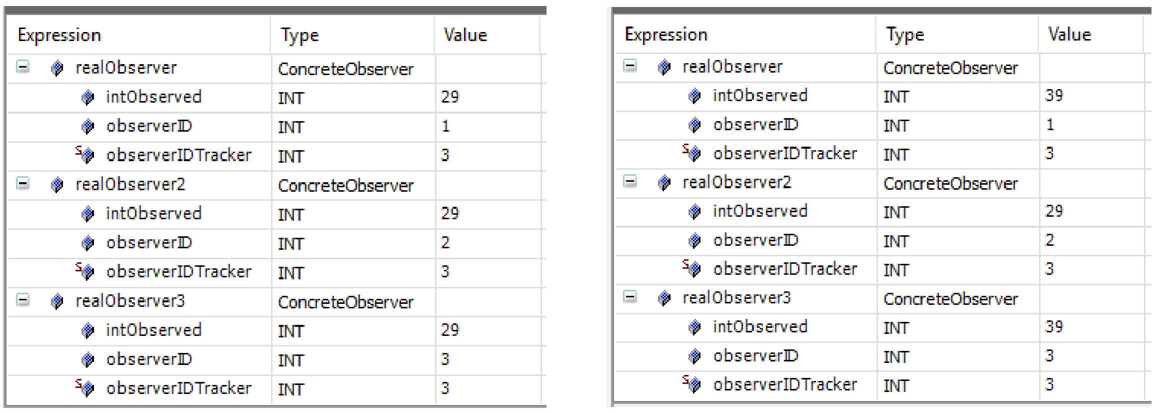
The code template for this pattern was developed following the class diagram described in the last section. It includes the ConcreteSubject POU, the ConcreteObserver POU, the Subject_Interface POU and the Observer_Interface POU as described in the UML class diagram.

The code template PLC_PRG, the client, considers one ConcreteSubject object and three ConcreteObserver instances.

After the registration of the every ConcreteObserver object, the PLC_PRG sends different Set request to the ConcreteSubject. Then the concrete subject notifies every subscribed observer of the changes.

```
    //Suscribe every observer(1, 2 and 3) to the subject
        realSubject . registerObserver ( realObserver ) ;
        realSubject . registerObserver ( realObserver2 ) ;
        realSubject . registerObserver ( realObserver3 ) ;

    //Changed on the observed value to be observed by the 3 observers
        realSubject . Set_observedINT ( 29 ) ;

    //Unsubsription of the observer 2
        realSubject . removeObserver ( realObserver2 ) ;

    //Changed on the observed value to be observed by the observers 1 and 3
        realSubject . Set_observedINT ( 39 ) ;
```

Figure 38: Client's request code



Expression	Type	Value
realObserver	ConcreteObserver	
intObserved	INT	29
observerID	INT	1
observerIDTracker	INT	3
realObserver2	ConcreteObserver	
intObserved	INT	29
observerID	INT	2
observerIDTracker	INT	3
realObserver3	ConcreteObserver	
intObserved	INT	29
observerID	INT	3
observerIDTracker	INT	3

(a) All observers are subscribed

Expression	Type	Value
realObserver	ConcreteObserver	
intObserved	INT	39
observerID	INT	1
observerIDTracker	INT	3
realObserver2	ConcreteObserver	
intObserved	INT	29
observerID	INT	2
observerIDTracker	INT	3
realObserver3	ConcreteObserver	
intObserved	INT	39
observerID	INT	3
observerIDTracker	INT	3

(b) Observer 2 is no longer subscribed

Figure 39: Observer code template sample results

The figure: 39 shows the different behaviour the ConcreteObservers have when they are subscribed or not. Using the debugging options in CODESYS, the developer can test the code step by step; the Figure: 38 shows a fraction of the PLC_PRG implementation used as test. On the figure 39a one can see how all the ConcreteObservers are subscribed and receive the notification from the ConcreteSubject. Figure 39b show the behaviour after the ConcreteObserver2, named realObserver2, was removed and another Set request was made to the ConcreteSubject object.

The Table 7 displays the overall results of the application after 700000 cycles. For the testing environment we are comparing the different results between a simulation environment in CODESYS and the real hardware running on a Raspberry Pi 3 model B+ with the CODESYS Control for Raspberry Pi SL runtime.

Table 7: Overall results of the Observer code template application

Description	Simulation	Raspberry Pi
Allocated memory size	94KB	133KB
Base cycle time	10ms	10ms
Average cycle time	15us	63us
Maximum cycle time	1296us	282us
Maximum Jitter	10573us	204us

The full code project can be found in section 6.7 Appendix G given in the Documentation format directly from the CODESYS environment.

3.3.3 Industrial application

As an industrial application, a Master HMI in charge of changing the observed value as well as registration and deregistration of observers is proposed.

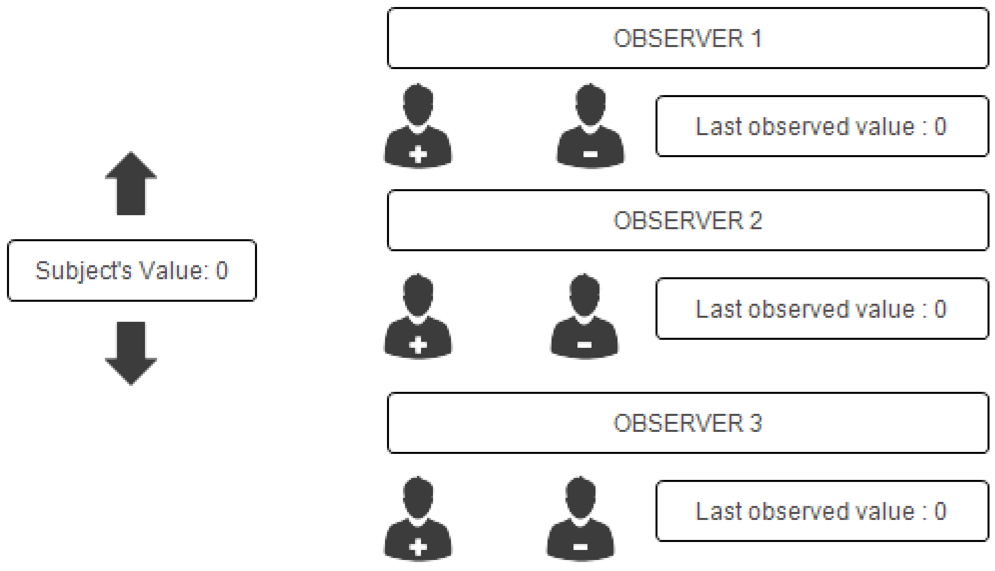


Figure 40: Observer application HMI

This HMI template, Figure: 40, replaces the PLC_PRG as the client. It includes the controls for increasing and decreasing the subject's observed value. Three observers are displayed with its controls to register and deregister from the subject and a display showing the last observed value from the subject.

As soon as the subject's observed value is changed, the automatic notification of the observers is triggered and displayed on each subscribed observer.

To complete the implementation, a few changes are considered in the class diagram Figure:41.

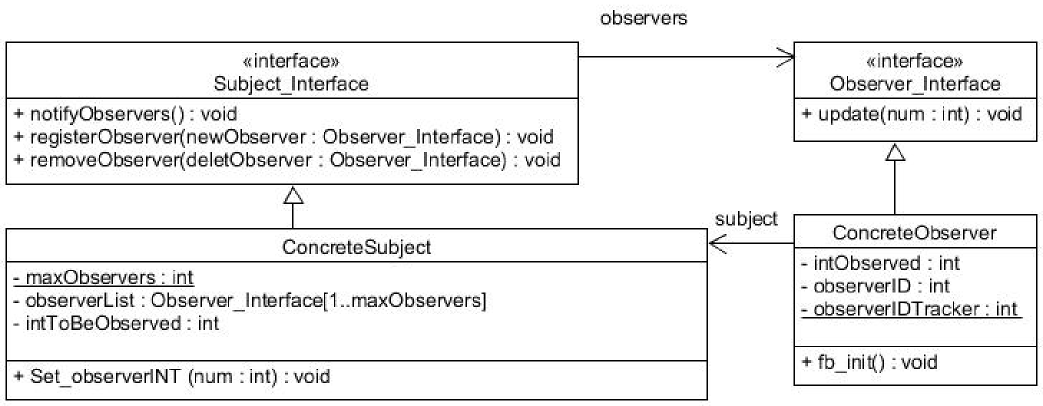

observers
«interface»
Subject_Interface
+ notifyObservers() : void
+ registerObserver(newObserver : Observer_Interface) : void
+ removeObserver(deletObserver : Observer_Interface) : void
«interface»
Observer_Interface
+ update(num : int) : void
subject
ConcreteSubject
- maxObservers : int
- observerList : Observer_Interface[1..maxObservers]
- intToBeObserved : int
+ Set_observerINT (num : int) : void
ConcreteObserver
- intObserved : int
- observerID : int
- observerIDTracker : int
+ fb_init() : void

Figure 41: Observer application class diagram

- ConcreteObserver:
  - Method *fb_init (): void*; The constructor method is responsible of the correct creation of the ConcreteObserver instance. It handles the correct use of both variables *observerID* and *observerIDTraker* at instantiation.

The Table 8 displays the overall results of the application with the same specifications of the last test. As most of the time the application will be on idle, the average cycle time in the simulation environment can be neglected.

Table 8: Overall results of the Observer industrial application

Description	Simulation	Raspberry Pi
Allocated memory size	1267KB	1788KB
Base cycle time	10ms	10ms
Average cycle time	n/a	44us
Maximum cycle time	401us	224us
Maximum Jitter	12026us	163us

The full code project can be found in section 6.8 Appendix H given in the Documentation format directly from the CODESYS environment.

3.4 Proxy design pattern

The Object Oriented capabilities of the IEC 61131-3 standard implemented in the CODESYS IDE, grant the developer of a basic concept of encapsulation. This can be done with the use of Program Organization Unit (POU) Properties object. These properties object can be configured to work as a getter, setter or both depending on the desired access mechanism a variable has. That, does not fulfil the security requirements in automation systems due to the lack of private variables.

In any system, there are certain variables that the client should not be able to modify or have access to. Thus, to enhance the data hiding mechanism of the software, the implementation of a Proxy design pattern is proposed. This Protection Proxy pattern is used as an access control mechanism in charge of filtering the request a certain object may receive, only passing those requests with the proper access rights to the object/field in question.

3.4.1 UML modeling

In this chapter, a simple Proxy Pattern will be described with UML notation.

To start with, an activity diagram will show the possible scenarios of a request being sent to the proxy object.

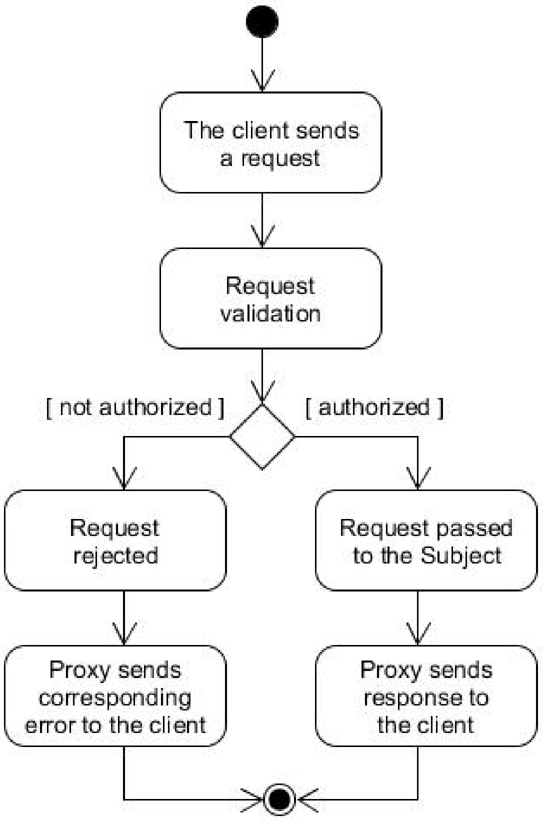

The client sends a request
Request validation
[not authorized]
[authorized]
Request rejected
Request passed to the Subject
Proxy sends corresponding error to the client
Proxy sends response to the client

Figure 42: Proxy activity diagram

From the activity diagram, a state and sequence diagrams of the request are proposed.

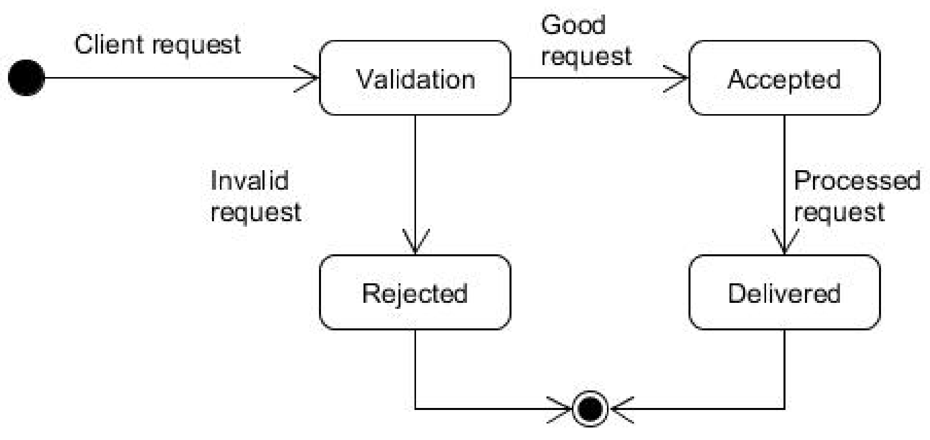


Figure 43: Proxy state diagram

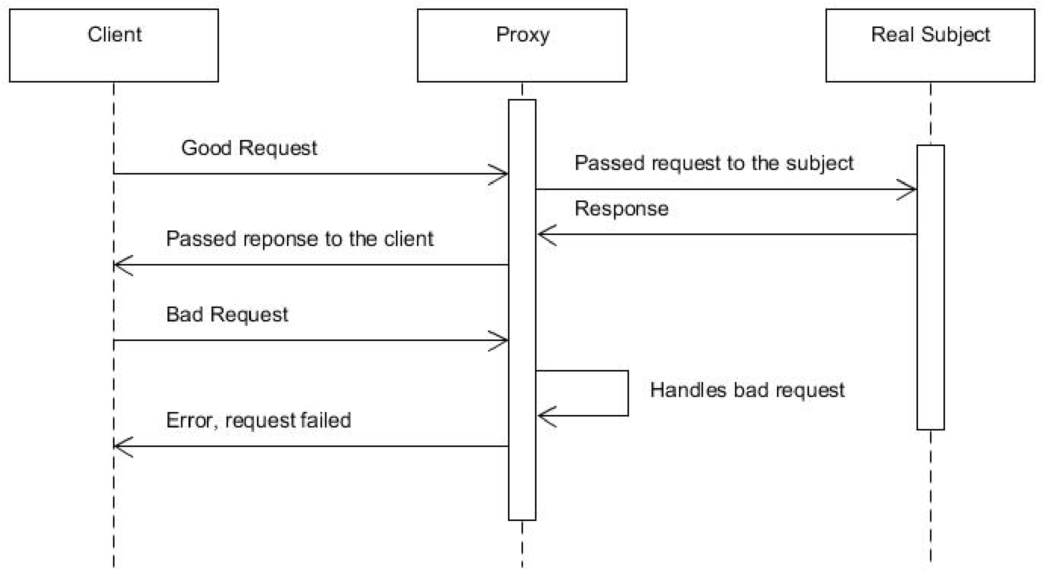


Figure 44: Proxy sequence diagram

As shown in the Figure: 44, when the Clients sends a "Good Request" to the proxy instance, it should be passed to the RealSubject instance. Either keeping its integrity or with added functionality by the proxy object. Then the proxy instance can forward the RealSubject's response to the client. In this case, again, the response can be passed with or without change by the proxy object.

In the case of a "Bad Request" by the client, the proxy instance should take full responsibility of the request and forward a reply to the client.

Finally, the class diagram of a protective Proxy, to be implemented in the CODESYS IDE, is presented and its basic functionality is also described.

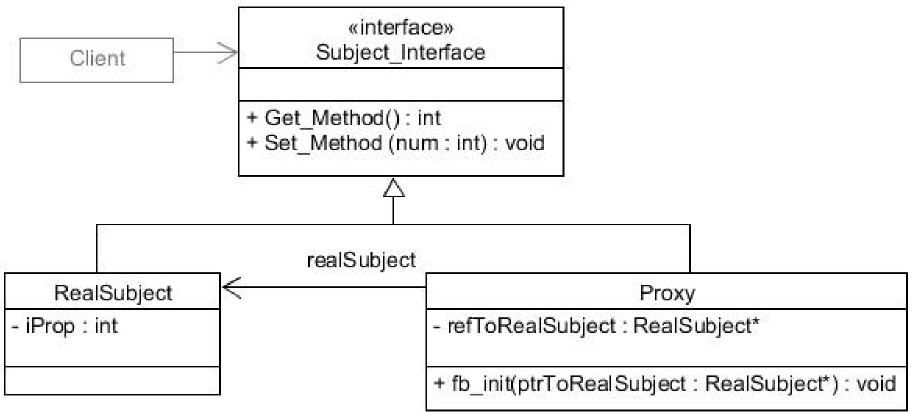

Client
«interface»
Subject_Interface
+ Get_Method() : int
+ Set_Method (num : int) : void
realSubject
RealSubject
- iProp : int
Proxy
- refToRealSubject : RealSubject*
+ fb_init(ptrToRealSubject : RealSubject*) : void

Figure 45: Proxy template class diagram

- Proxy:
  - Variable *refToRealSubject : RealSubject*; This variable is defined as a pointer to the RealSubject class. It will hold the address reference to the concrete RealSubject instance that the proxy object is working on.
  - Method *fb_ init (ptrToRealSubject : RealSubject*): void*; The constructor method is responsible of the correct creation of the proxy instance and write the passed RealSubject* to the proxy's own field refToRealSubject. The parameter *ptrTo-RealSubject* is the current address of the RealSubject instance, to be protected by the proxy.
- Subject_Interface:
  - Method *Get_ Method() : int*; This method allows the read option of an integer.
  - Method *Set_ Method (num : int) : void*; This method allows the write option of an integer field to be written with the passed value of the parameter *num*.
- RealSubject:
  - Variable *iProp : int*; Integer variable to be modified by the proxy instance if the client's request is accepted by the proxy instance.

Following this defined software architecture, a domain specific Proxy pattern code template is proposed.

3.4.2 Code template

The implementations was performed in the CODESYS IDE with the specific runtime: CODESYS Control for Raspberry Pi SL.

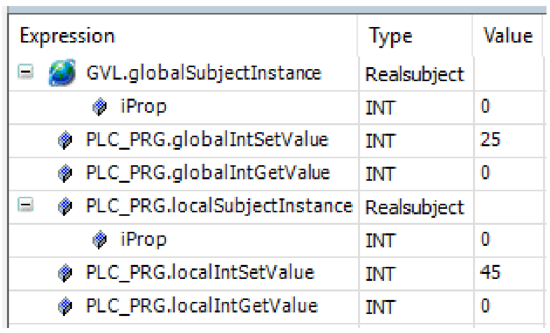
The code template for this pattern was developed following the class diagram described in the last section. It includes the Proxy POU, the RealSubject POU and the Subject_Interface POU as described in the UML class diagram.

The code template PLC_PRG, the client, considers two RealSubject objects and one Proxy pointer on different scope spaces.

The proxy object is dynamically created by the proxy pointer and using a pointer to the RealSubject object as an input parameter, to hold the reference to the object to be modified. Once the request for that specific RealSubject object is completed, the proxy object is deleted to free the memory and resource so that another RealSubject object can instantiate the proxy and realize a connection. The full interaction between the client, proxy and RealSubject objects is shown in Figure:46.

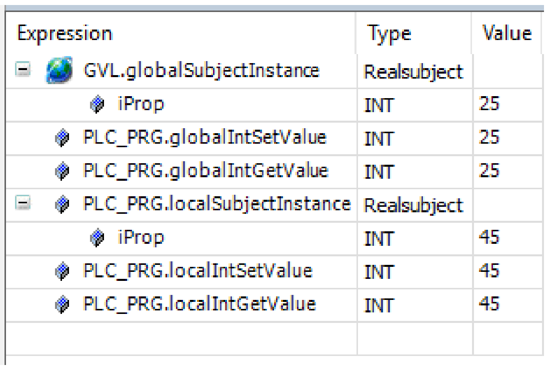
```
//The PLC_PRG client will handle Set and Get request to both RealSubject
instances with the same Proxy object

//First we verify if the pointer to RealSubject to be used is a valid pointer
IF ptrLocalSubject  <>  ptrNullRealSubject
    //If so,
    THEN
    //Dynamically creates a Proxy object using the Proxy pointer and the
RealSubject verified pointer
    ptrProxy  :=  __NEW ( Proxy ( ptrToRealSubject  :=  ptrLocalSubject ) ) ;
    //The Proxy object makes the Set request and writes to the RealSubject
object
    ptrProxy ^ . Set_Method ( localIntSetValue ) ;
    //A Get request is made to the Proxy object and it passes the request to the
 RealSubject and in the end delivers the result to the client
    localIntGetValue  :=  ptrProxy ^ . Get_Method ( ) ;
    //Once the requests are handled, the Proxy object is deleted to free memory
and to free the Proxy pointer resource
    __DELETE ( ptrProxy ) ;
END_IF

```

Figure 46: Client's request code



Expression	Type	Value
GVL.globalSubjectInstance	Realsubject	
iProp	INT	0
PLC_PRG.globalIntSetValue	INT	25
PLC_PRG.globalIntGetValue	INT	0
PLC_PRG.localSubjectInstance	Realsubject	
iProp	INT	0
PLC_PRG.localIntSetValue	INT	45
PLC_PRG.localIntGetValue	INT	0

(a) Values for the Realsubject objects before one RUN cycle



Expression	Type	Value
GVL.globalSubjectInstance	Realsubject	
iProp	INT	25
PLC_PRG.globalIntSetValue	INT	25
PLC_PRG.globalIntGetValue	INT	25
PLC_PRG.localSubjectInstance	Realsubject	
iProp	INT	45
PLC_PRG.localIntSetValue	INT	45
PLC_PRG.localIntGetValue	INT	45

(b) Values for the Realsubject objects after one RUN cycle

Figure 47: Proxy code template sample results

As is noted in figure: 47, after one RUN cycle the value of the iProp field on both Real-Subjec objects has changed. The PLC_PRG has made a set request to both RealSubject objects through the proxy object. The specific set request were done with the specific values in IntSetValue for each RealSubject and the PLC_PRG makes a get request to validate the change on the iProp field of both RealSubject objects.

The Table 9 displays the overall results of the application after 700000 cycles. For the testing environment we are comparing the different results between a simulation environment in CODESYS and the real hardware running on a Raspberry Pi 3 model B+ with the CODESYS Control for Raspberry Pi SL runtime.

Table 9: Overall results of the Proxy code template application

Description	Simulation	Raspberry Pi
Allocated memory size	89KB	127KB
Base cycle time	10ms	10ms
Average cycle time	23us	58us
Maximum cycle time	2281us	259us
Maximum Jitter	15847us	272us

The full code project can be found in section 6.9 Appendix I, given in the Documentation format directly from the CODESYS environment.

3.4.3 Industrial application

As an example of an industrial application, a Master HMI, in charge of changing the subject's value (INT) is proposed as shown in Figure: 48.

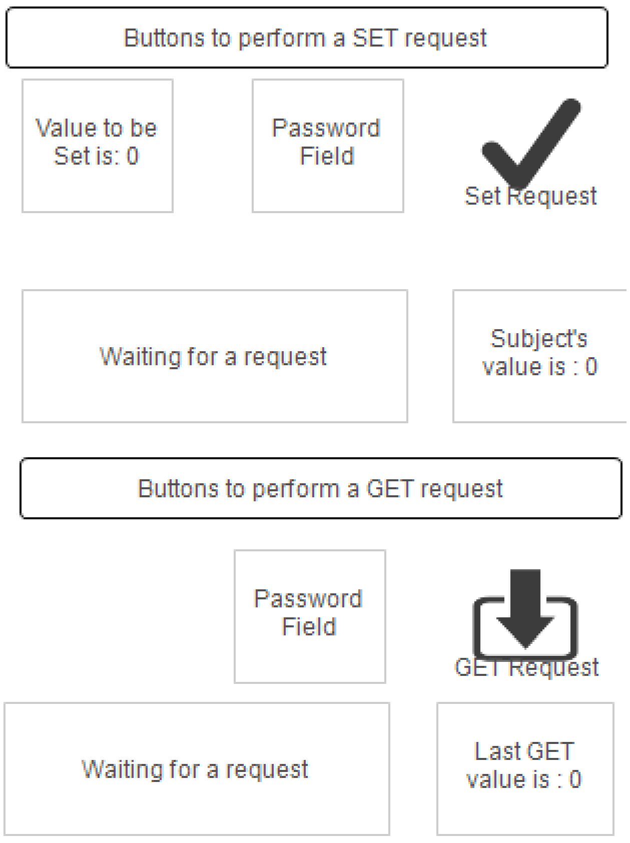


Figure 48: Proxy application HMI

This HMI template replaces the PLC_PRG as the client. It can make either set or get requests to the PLC.

The HMI includes the controls to specify, an integer value to be set to the subject, change the password for the specific request and the set and get request buttons. The password field is used to demonstrate, how the request must fulfil the access mechanism, in order to be passed by the proxy object. This password field is cleared after every request as a security mechanism.

Once a request is sent to the PLC_PRG, the proxy object will handle the request. The proxy object can either accept the request and pass it to the subject to be handled properly or reject the request and notify the client (HMI) about the rejection.

To complete the implementation, a few changes are considered in the class diagram Figure:49.

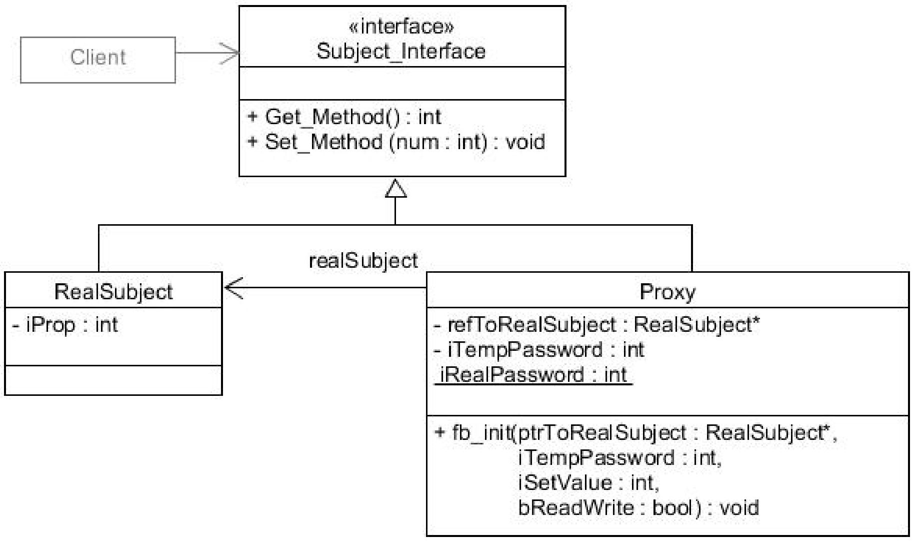

Client
«interface»
Subject_Interface
+ Get_Method() : int
+ Set_Method (num : int) : void
realSubject
RealSubject
- iProp : int
Proxy
- refToRealSubject : RealSubject*
- iTempPassword : int
iRealPassword : int
+ fb_init(ptrToRealSubject : RealSubject*, iTempPassword : int, iSetValue : int, bReadWrite : bool) : void

Figure 49: Proxy application class diagram

- Proxy:
  - Variable *iTempPassword : int*; This input variable holds the passed value of the client's current password field, to be verified by the proxy instance.
  - Variable *iRealPassword : int*; This static variable holds the actual password code.
  - Variable *iSetValue : int*; This variable holds the value to be set by the proxy instance if the request passes the security mechanism.
  - Variable *bReadWrite : bool*; This boolean variable is used as identifier of the request. Either ser (boolean value TRUE) or get (boolean value FALSE).

The Table 10 displays the overall results of the application with the same specifications of the last test. As most of the time the application will be on idle, the average cycle time in the simulation environment can be neglected.

Table 10: Overall results of the Proxy industrial application

Description	Simulation	Raspberry Pi
Allocated memory size	1410KB	2026KB
Base cycle time	10ms	10ms
Average cycle time	n/a	49us
Maximum cycle time	641us	142us
Maximum Jitter	8444us	65us

The full code project can be found in section 6.10 Appendix J given in the Documentation format directly from the CODESYS environment.

3.5 Singleton design pattern

The Singleton design pattern has some special characteristics and should be handled differently.

A classic Java implementation of the Singleton pattern will make it easier to understand and visualize the problems on the CODESYS environment at compile time. This implementations uses "Lazy Instanciation" to make it simpler.

```
1 public class Singleton {
2   //A static field of type Singleton to store the one and only Singleton
3 //instance
4 private static final Singleton INSTANCE = new Singleton();
5
6 //The private constructor restricts the object's instanciation to
7 //the class only
8   private Singleton() {}
9
10  //Static method to get access to the sole Singleton instance
11 public static Singleton getInstance() {
12
13 //The lazy instanciation manages the creation of the instance
14      //only if it is needed.
15      if (INSTANCE == null) {
16       INSTANCE = new Singleton();
17    }
18
19  //The the method will grant access to the Singleton INSTANCE
20 return INSTANCE;
21
22 }
23
24 }
```

The problems to make a classic Singleton pattern implementation on CODESYS and IEC 61131-3 are:

1. The constructor *fb_init*(); in CODESYS can not be private; This leaves the instanciation process unrestricted and many objects could be declared of the Singleton class.

2. There are no static methods in CODESYS; A static method in JAVA can be accessed without an instance. One can just use the name of the class and use the static method *Singleton.getInstance*(); in this case but in CODESYS all the methods from a class must be called from an instance of that class.

3. Data recursion is not yet implemented in CODESYS; In line 4 of the JAVA example, a case of data recursion will occur. This will lead to error at compile time.

4. The THIS operator is not a valid as assignment target; This means that the calling object can not modify its own address to point to the first Singleton instance.

Because of this problems, the classic Singleton implementation can not be done. A new implementation is proposed usign a proxy object in between the client and the singleton class.

The proxy will solve the problem in the following way:

1. The proxy object will handle the dynamic creation of an instance of the singleton class; The constructor of the singleton class will still be public but if the creation of the instances of that class is restricted through a proxy, the proxy object can then decide whether or not create more that one instance of the singleton class.
2. The created singleton instance will be stored in a static field on the proxy object, that way, even if there are more than one proxy objects they will all share the same singleton instance in one of their fields. This way the static methods, data recursion and the THIS operator problems are solved.

3.5.1 UML modeling

In this chapter, a modified Singleton Pattern will be described with UML notation.

To start with, an activity diagram will show the scenario where a modified Singleton pattern is needed.

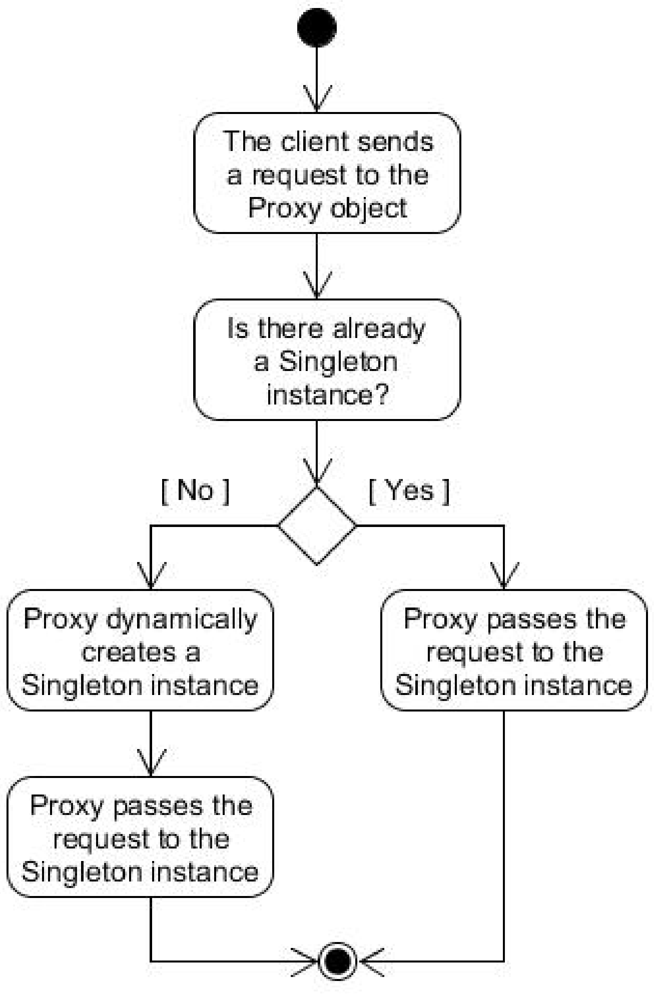

The client sends a request to the Proxy object
Is there already a Singleton instance?
[No]
[Yes]
Proxy dynamically creates a Singleton instance
Proxy passes the request to the Singleton instance
Proxy passes the request to the Singleton instance

Figure 50: Singleton activity diagram

From the activity diagram, a state diagram of the request is proposed.

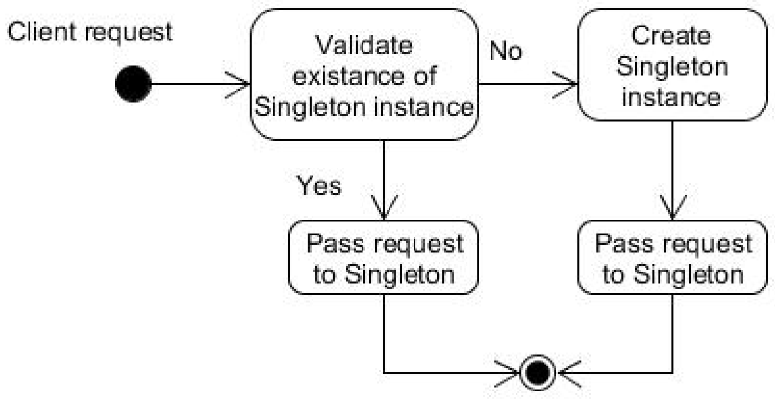

Client request
Validate existance of Singleton instance
No
Create Singleton instance
Yes
Pass request to Singleton
Pass request to Singleton

Figure 51: Singleton state diagram

The sequence diagram to handle all scenarios of the client's request is proposed.

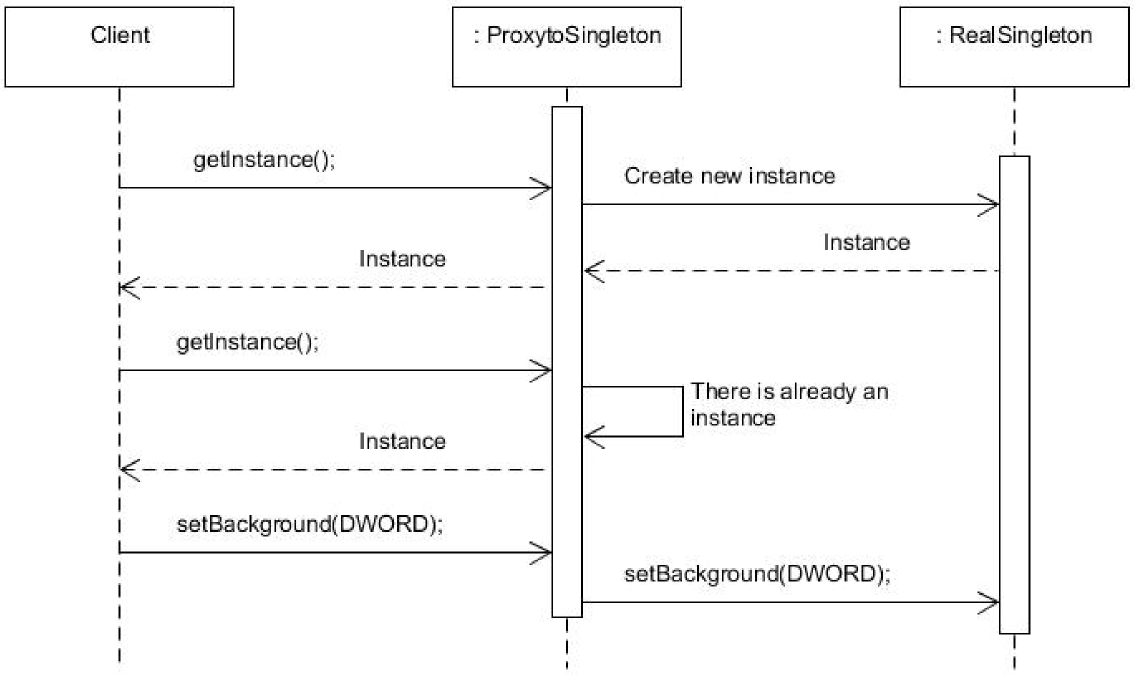


Figure 52: Singleton sequence diagram

As shown in the Figure: 52, the proxy object has to handle every request from the client. If the client request the creation of a singleton object, the proxy must either create the first singleton instance or return the same first instance. If the request from the client does not involve the creation of a singleton instance, the proxy object can simply forward the request to the singleton instance.

Finally, the class diagram of a modified Singleton pattern, to be implemented in the CODESYS IDE, is presented and its basic functionality is also described.

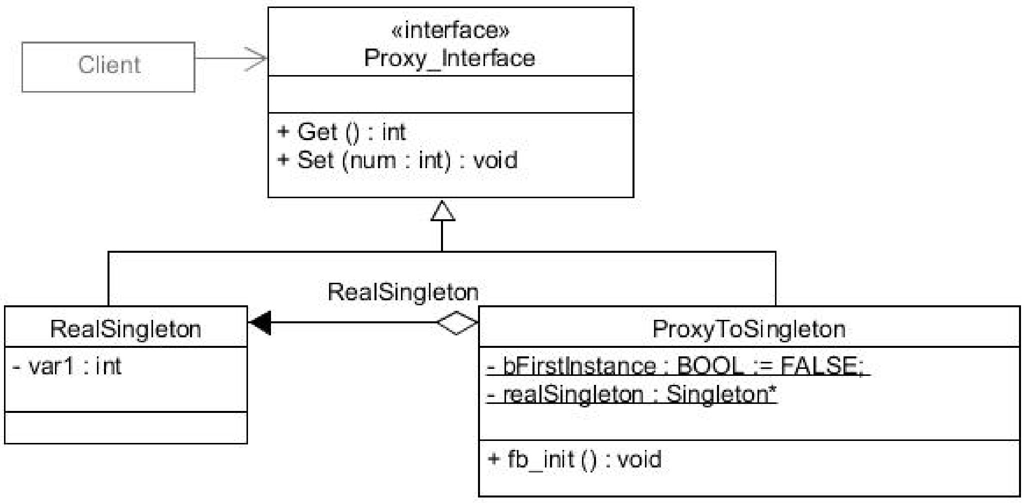


Figure 53: Singleton template class diagram

- ProxyToSingleton:
  - Static variable *bFirstInstance : BOOL := FALSE*; This variable is defined as static to be available to all Proxy instances. Its basic responsibility is to hold the reference to the creation of the singleton instance. This means, it knows the status of whether or not a singleton instances has been created.
  - Static variable *realSingleton : Singleton**; This static variable, is declared as as pointer to an instance of the singleton FB. That way, this pointer is the only declared instance of singleton class in the program. It can be dynamically created by the first proxy instance and then, the constructor should avoid a duplicate of the singleton instance based on the *bFirstInstance* current value.
  - Method *fb_init() : void* In this case, the constructor is responsible of the one and only creation of the singleton instance.
- Subject_Interface:
  - Method *Get_() : int*; This method allows the read option of an integer.
  - Method *Set_(num : int) : void*; This method allows the write option of an integer field to be written with the passed value of the parameter *num*.
- RealSubject:
  - Variable *var1 : int*; Integer variable to be modified by the proxy instance if the client's request is accepted by the proxy instance.

Following this defined software architecture, a domain specific Singleton pattern code template is proposed.

3.5.2 Code template

The implementations was performed in the CODESYS IDE with the specific runtime: CODESYS Control for Raspberry Pi SL.

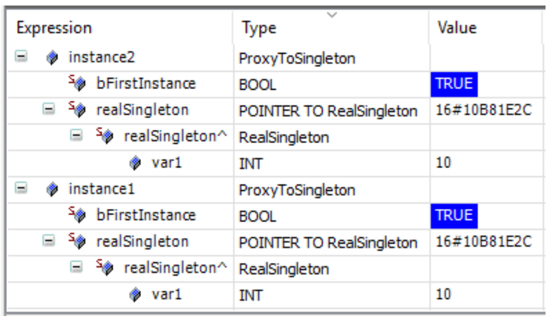
The code template for this pattern was developed following the class diagram described in the last section. It includes the ProxyToSingleton POU, the RealSingleton POU and the Proxy_Interface POU as described in the UML class diagram.

The code template PLC_PRG, the client, considers two ProxyToSingleton objects and one RealSingleton object. There are also two pointers to RealSingleton to verify that the two ProxyToSingleton objects in its static realSingleton field have the same object.

As previously mentioned, the first ProxyToSingleton object is in charge of the dynamic creation of the RealSingleton object, once the RealSingleton object is created, every ProxyToSingleton will have access to the only RealSingleton object, so it will not matter which of the two ProxyToSingleton objects make the request, the RealSingleton object will always receive the request as shown in Figures: 54 and 55.

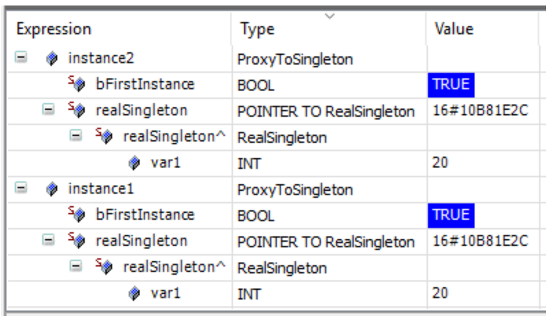
```
//We verify that the instance1 has Read and Write access to the RealSingleton instance
instance1 . Set ( 10 ) ;
iGetValue  :=  instance1 . Get ( ) ;

//We verify that the instance2 has Read and Write access to the RealSingleton instance
instance2 . Set ( 20 ) ;
iGetValue := instance2 . Get ( ) ;
```

Figure 54: Client's request code



Expression	Type	Value
instance2	ProxyToSingleton	
bFirstInstance	BOOL	TRUE
realSingleton	POINTER TO RealSingleton	16#10B81E2C
realSingleton^	RealSingleton	
var1	INT	10
instance1	ProxyToSingleton	
bFirstInstance	BOOL	TRUE
realSingleton	POINTER TO RealSingleton	16#10B81E2C
realSingleton^	RealSingleton	
var1	INT	10

(a) Values of the RealSingleton object after a request from the instance1 is made



Expression	Type	Value
instance2	ProxyToSingleton	
bFirstInstance	BOOL	TRUE
realSingleton	POINTER TO RealSingleton	16#10B81E2C
realSingleton^	RealSingleton	
var1	INT	20
instance1	ProxyToSingleton	
bFirstInstance	BOOL	TRUE
realSingleton	POINTER TO RealSingleton	16#10B81E2C
realSingleton^	RealSingleton	
var1	INT	20

(b) Values of the RealSingleton object after a request from the instance2 is made

Figure 55: Singleton code template sample results

As is noted in figure: 55, both ProxyToSingleton instances have the same address in its pointer to RealSubject field, so both have access to the same RealSingleton object and a request made to any of the ProxyToSingleton objects will alter the results on the RealSingleton object.

The Table 11 displays the overall results of the application after 700000 cycles. For the testing environment we are comparing the different results between a simulation environment in CODESYS and the real hardware running on a Raspberry Pi 3 model B+ with the CODESYS Control for Raspberry Pi SL runtime.

Table 11: Overall results of the Singleton code template application

Description	Simulation	Raspberry Pi
Allocated memory size	88KB	127KB
Base cycle time	10ms	10ms
Average cycle time	15us	58us
Maximum cycle time	1178us	259us
Maximum Jitter	39912us	272us

The full code project can be found in section 6.9 Appendix I given in the Documentation format directly from the CODESYS environment.

3.5.3 Industrial application

As an industrial application, a Master HMI displaying the default background stored in a configuration file (Singleton instance) is proposed.

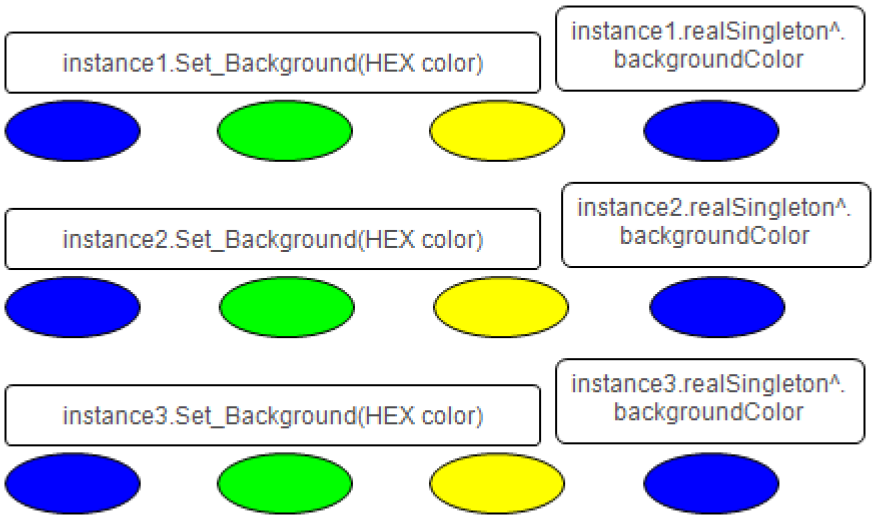

instance1.Set_Background(HEX color)
instance1.realSingleton^. backgroundColor
instance2.Set_Background(HEX color)
instance2.realSingleton^. backgroundColor
instance3.Set_Background(HEX color)
instance3.realSingleton^. backgroundColor

Figure 56: Singleton application HMI

This HIM, Figure: 56, template replaces the PLC_PRG as the client. It can make Set requests and by default makes the Get request to the background colour.

In the HMI there are three proxy instances that are able to change the values in the singleton instance. It does not matter which proxy instance triggers the background change, in the end all proxy instances receive the new background because they are modifying the same singleton instance.

To complete the implementation, a few changes are considered in the next class diagram

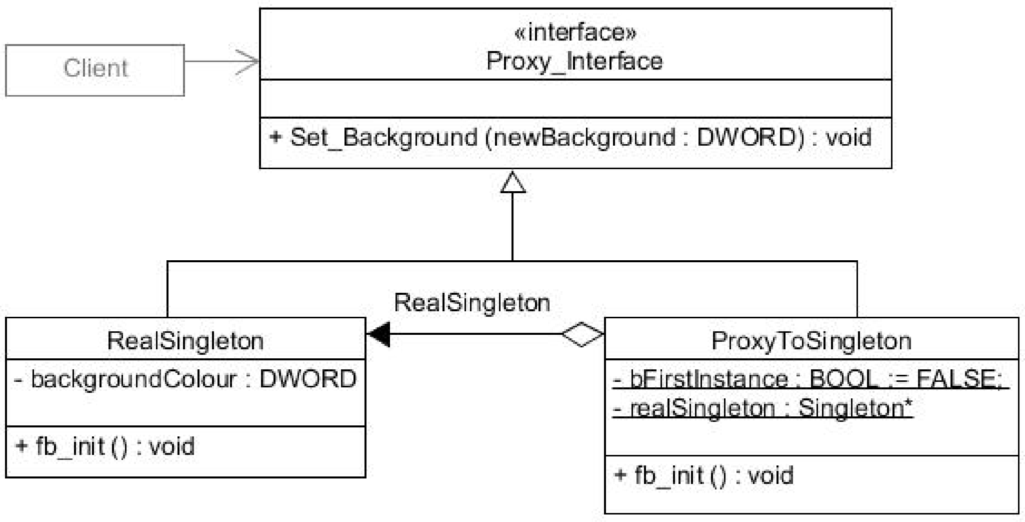

Client
«interface»
Proxy_Interface
+ Set_Background (newBackground : DWORD) : void
RealSingleton
RealSingleton
- backgroundColour : DWORD
+ fb_init () : void
ProxyToSingleton
- bFirstInstance : BOOL := FALSE;
- realSingleton : Singleton*
+ fb_init () : void

Figure 57: Singleton application class diagram

- Proxy_Interface:
  - Method *Set_Background (newBackground : DWORD) : void*; This method grants the functionality, to both the RealSingleton FB and the ProxyToSingleton FB, of being able to modify the backgroundColour variable in the RealSingleton instance.
  - Variable *newBackground : DWORD*; This variable is passed to modify the value inside the backgroundColour field in the RealSingleton instance.

- RealSingleton:
  - Variable *backgroundColour : DWORD*; This variable holds the current background colour of the application and can be modified only by the ProxyToSingleton instances.
  - Method *fb_init() : void*; The constructor method of the RealSingleton FB set the default background colour to blue.

The Table 12 displays the overall results of the application with the same specifications of the last test.

Table 12: Overall results of the Proxy industrial application

Description	Simulation	Raspberry Pi
Allocated memory size	1250KB	2026KB
Base cycle time	10ms	10ms
Average cycle time	17us	49us
Maximum cycle time	714us	142us
Maximum Jitter	10606us	65us

The full code project can be found in section 6.12 Appendix L given in the Documentation format directly from the CODESYS environment.

4 Conclusions

The growing complexity of automation systems demanded a change. New techniques and methods are required in the industry to address its needs. These techniques must be domain specific in order to be useful to the automation industry.

The Codesys IDE is a great tool for prototyping new concepts. The simulation and visualization options can be used for testing these concepts, that can be directly applied and deployed in the industry without being target specific; the developer can write the whole architecture and behaviour of the program and then, map the I/O from the chosen target to the already existing variables. This way, if the hardware does not exist any more, the core of the system remains intact and switching the hardware, is a much easier task than developing the project again.

The concept of object oriented programming is relatively new in the automation industry but, it is a well known concept for systems engineers, in fact, it has been studied since 1962. With the latest update to the IEC 6113-3 standard and the introduction of the OOP basic concepts for software development of PLC based systems, a wide door of possibilities has been open for automation engineers, to exploit and make use of it.

In the autor's opinion, the wisest move in software development for embedded systems programmed under the IEC 6113-3 standard; is to learn, adopt and apply the more than 50 years of effort and research in software development for PC based systems. This ideology could reduce the time gap and research effort of automation software developers, researchers and educators by having 50 years of continuous research and knowledge from the PC based systems. One clear example of this are design patterns.

A design pattern is the use of a defined architecture, that has been tried and tested as a proven solution to many common recurring problems. These reusable solutions are a set of rule of conventions. These rules, include a list of objects used in the application and an effective description of how these objects interrelate and interact with each other, this means patterns not only describe how software is structured, but more importantly, they also describe how classes and objects interact, especially at run time. Other important advantages of design patterns are:

- Reduction of the technical risk of deploying a new and untested design.
- Improvement of code readability.
- Reusability and extensibility of the already developed applications.
- Reduction of development time and thus reduction of the final cost of the system.
- Easier implementation of principles of good design.
- Easier understanding of the documentation.
- Easier communication between developers.

Design patterns are a well understood and documented resource in the PC based systems field but, they are not deterministic or prescriptive solutions; Rather they are abstract solutions that can be tailored to the specific problem in hand. In our case, they have to be adopted to PLC based systems to solve domain specific problems in software development.

The main reason behind the possible adoption of Design patterns is that they are language neutral, so can be applied to any language that supports object-orientation.

Design Patterns are highly flexible and can be used in practically any type of application or domain.

This thesis is an adoption of design patterns for PC based systems to domain specific design patterns for the automation industry. The explored design patterns explained in this thesis are, the Builder pattern (both classical and optional approaches), Decorator pattern, Observer pattern, Proxy pattern and a workaround implementation of the Singleton pattern.

A summary for this specific work is:

- The common use for each of the proposed design patterns was explained.
- The creation of domain specific UML models for software development of each proposed design pattern was made.
- The generation of code template from these domain specific UML models was developed.
- An industrial application to be solved with the domain specific design patterns template was proposed.
- Implementation and simulation of the proposed industrial applications, with aid of visualization elements, was developed and tested in a PLC based IDE and on a Raspberry Pi 3.

For future work, there are more concepts on the PC based systems, that could be adopted to PLC based systems. Some examples of this are; object oriented databases, concurrent programming and a workaround recursive programming (similar to the adjustment to the Singleton pattern). As PLC's are getting more potent and robust, these possibilities are not very far away from being executed.

Another direction for future work, besides the adoption of more design patterns, is a virtual plant simulation integrating more than one design pattern to control and fulfil the requirements of an automation plant as a whole.

5 References

[1] D. Witsch and B. Vogel-Heuser. *Close integration between UML and IEC61131-3:New possibilities through object-oriented extensions.* in Proc. IEEE Conf. Emerging Technol. Factory Autom., 2009, pp. 1–6.

[2] B. Werner. *Object-oriented extensions for IEC 61131-3.* IEEE Ind. Electron. Mag., vol. 3, no. 4, pp. 36–39, Dec. 2009.

[3] V. Vyatkin. *Software Engineering in Industrial Automation: State-of-the-Art Review.* Industrial Informatics, IEEE Transactions on, vol. 9, no. 3, pp. 1234–1249, Aug 2013.

[4] D. Tikhonov, D. Schütz, S. Ulewicz and B. Vogel-Heuser. *Towards industrial application of model-driven platform-independent PLC programming using UML.* IECON 2014 - 40th Annual Conference of the IEEE Industrial Electronics Society, Dallas, TX, 2014, pp. 2638-2644.

[5] F. Serna, C. Catalan, A. Blesa, and J. M. Rams. *Design patterns for failure management in IEC 61499 function blocks.* in Proc. IEEE Conf. Emerging Technol. Factory Autom., 2010, pp. 1–7.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[7] V. Gonzandez , A. S. Diaz , P. G. Fernandez , A. F. Junquera and R. M. Bayon. *MIOOP. An object oriented programmingparadigm approach on the IEC 61131 standard.* Proc.IEEE Conf. Emerging Technol. Factory Autom. (ETFA 2010), pp. 1-4, 2010

[8] 3S SmartSoftware Solutions. *Codesys v3* 2012., [online] Available: http://www.3s-software.com

[9] N. Papakonstantinou, S. Sierla, and K. Koskinen. *Object oriented extensions of IEC 61131-3 as an enabling technology of software product lines.* in Proc. 16th IEEE Conf. Emerg. Technol. Fact. Autom. (ETFA), Toulouse, France, 2011, pp. 1–8.

[10] F. Basile, P. Chiacchio and D. Gerbasio. *On the Implementation of Industrial Automation Systems Based on PLC.* in IEEE Transactions on Automation Science and Engineering, vol. 10, no. 4, pp. 990-1003, Oct. 2013.

[11] M. Bonfe and C. Fantuzzi. *A practical approach to object-oriented modeling of logic control systems for industrial applications.* Decision and Control, 2004. CDC. 43rd IEEE Conference on, Nassau, 2004, pp. 980-985 Vol.1.

[12] C. Secchi, M. Bonfe, C. Fantuzzi, R. Borsari and D. Borghi. *Object-Oriented Modeling of Complex Mechatronic Components for the Manufacturing Industry.* in IEEE/ASME Transactions on Mechatronics, vol. 12, no. 6, pp. 696-702, Dec. 2007.

[13] K. H. Han and J. Jeon. *Object-oriented design and simulation of automated manufacturing system.* Information Science, Electronics and Electrical Engineering (ISEEE), 2014 International Conference on, Sapporo, 2014, pp. 498-502.

[14] M. Pineda-Sanchez et al. *Programmable Logic Controllers (PLC) in the packaging industry: An object oriented approach for developing control programs.* 2014 9th International Microsystems, Packaging, Assembly and Circuits Technology Conference (IMPACT), Taipei, 2014, pp. 386-389.

[15] A. M. Fernández-Sáez, D. Caivano, M. Genero and M. R. V. Chaudron. *On the use of UML documentation in software maintenance: Results from a survey in industry.* Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on, Ottawa, ON, 2015, pp. 292-301.

[16] K. Sacha. *Verification and Implementation of Dependable Controllers.* Dependability of Computer Systems, 2008. DepCos-RELCOMEX '08. Third International Conference on, Szklarska Poreba, 2008, pp. 143-151.

[17] L. Bassi, C. Secchi, M. Bonfe and C. Fantuzzi. *A SysML-Based Methodology for Manufacturing Machinery Modeling and Design.* in IEEE/ASME Transactions on Mechatronics, vol. 16, no. 6, pp. 1049-1062, Dec. 2011.

[18] N. Papakonstantinou and S. Sierla. *Generating an object oriented IEC 61131-3 software product line architecture from SysML.* in Proc. IEEE 18th Conf. Emerg. Technol. Fact. Autom. (ETFA), Cagliari, Italy, 2013, pp. 1–8.

[19] D. Witsch, M. Ricken, B. Kormann and B. Vogel-Heuser. *PLC-statecharts: An approach to integrate umlstatecharts in open-loop control engineering.* 2010 8th IEEE International Conference on Industrial Informatics, Osaka, 2010, pp. 915-920.

[20] D. Friedrich and B. Vogel-Heuser. *Benefit of system modeling in automation and control education.* 2007 American Control Conference, New York, NY, 2007, pp. 2497-2502.

[21] H. Dibowski, J. Ploennigs, and K. Kabitzsch. *Automated design of building automation systems.* IEEE Trans. Ind. Electron., vol. 57, no. 11, pp. 3606–3613, Nov. 2010.

[22] F. Serna, C. Catalan, A. Blesa, and J. M. Rams. *Design patterns for failure management in IEC 61499 function blocks.* in Proc. IEEE Conf. Emerging Technol. Factory Autom., 2010, pp. 1–7.

[23] V. Dubinin and V. Vyatkin. *Semantics-robust design patterns for IEC 61499.* IEEE Trans. Ind. Inf., vol. 8, no. 2, pp. 279–290, May 2012.

[24] F. Serna, C. Catalán, A. Blesa, J. M. Colom and J. M. Rams. *"Predictive maintenance surveyor" design pattern for machine tools control software applications.* Emerging Technologies and Factory Automation (ETFA), 2011 IEEE 16th Conference on, Toulouse, 2011, pp. 1-7.

[25] K. Eckert, A. Fay, T. Hadlich, C. Diedrich, T. Frank and B. Vogel-Heuser. *Design patterns for distributed automation systems with consideration of non-functional requirements.* Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies and Factory Automation (ETFA 2012), Krakow, 2012, pp. 1-9.

[26] M. Steinegger, A. Zoitl, M. Fein and G. Schitter. *Design patterns for separating fault handling from control code in discrete manufacturing systems.* Industrial Electronics Society, IECON 2013 - 39th Annual Conference of the IEEE, Vienna, 2013, pp. 4368-4373.

[27] W. Dai and V. Vyatkin. *A component-based design pattern for improving reusability of automation programs.* Industrial Electronics Society, IECON 2013 - 39th Annual Conference of the IEEE, Vienna, 2013, pp. 4328-4333.

[28] L. Racchetti, C. Fantuzzi, L. Tacconi and M. Bonfe. *The PLC UML State-chart design pattern.* Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA), Barcelona, 2014, pp. 1-4.

[29] M. Samek. *Practical Statecharts in C/C+: Quantum Programming for Embedded SystemscMiro.* Taylor & Francis US, 2002.

6 Appendices

6.1 Appendix A: Classic Builder pattern template

Please refer to the files AppendixA.pdf and AppendixA.project

6.2 Appendix B: Classic Builder pattern application

Please refer to the files AppendixB.pdf and AppendixB.project

6.3 Appendix C: Optional Builder pattern template

Please refer to the files AppendixC.pdf and AppendixC.project

6.4 Appendix D: Optional Builder pattern application

Please refer to the files AppendixD.pdf and AppendixD.project

6.5 Appendix E: Decorator pattern template

Please refer to the files AppendixE.pdf and AppendixE.project

6.6 Appendix F: Decorator pattern application

Please refer to the files AppendixF.pdf and AppendixF.project

6.7 Appendix G: Observer pattern template

Please refer to the files AppendixG.pdf and AppendiG.project

6.8 Appendix H: Observer pattern application

Please refer to the files AppendixH.pdf and AppendixH.project

6.9 Appendix I: Proxy pattern template

Please refer to the files AppendixI.pdf and AppendixI.project

6.10 Appendix J: Proxy pattern application

Please refer to the files AppendixJ.pdf and AppendixJ.project

6.11 Appendix K: Singleton pattern template

Please refer to the files AppendixK.pdf and AppendixK.project

6.12 Appendix L: Singleton pattern application

Please refer to the files AppendixL.pdf and AppendixL.project