UNIVERSITÀ DI PARMA

UNIVERSITA' DEGLI STUDI DI PARMA

Dottorato di Ricerca in Tecnologie dell'Informazione

XXXI Ciclo

Software and Firmware Design for an Embedded Framework Applied to Electric Drives

Coordinatore:

Chiar.mo Prof. Marco Locatelli

Tutor:

Chiar.mo Prof. Carlo Concari

Dottorando: *Roberto Zanichelli*

Anni 2015/2018

A chi mi vuole bene,
A chi mi ha insegnato,
A tutti quelli che mettono
il cuore in ciò che fanno

Sommario

Lo sviluppo di componenti firmware e software per la realizzazione di dispositivi elettronici embedded, utilizzati in ambito industriale, è un'attività che richiede tempo, sforzi e numerose competenze. La crescente complessità e l'evoluzione della tecnologia, hanno portato alla necessità di impiegare strumenti e strategie per lo sviluppo di programmi inerenti a questi prodotti, al fine di poter garantire un tempo ragionevole di rilascio del prodotto, affidabilità ed efficienza. Lo sviluppo di questi prodotti richiede diverse discipline e strumenti. I dispositivi embedded industriali sono spesso forniti assieme a programmi software per la parametrizzazione e il monitoraggio del dispositivo. Di solito comunicano con questi strumenti e altri dispositivi di supervisione, attraverso protocolli di comunicazione industriali. Questo tipo di dispositivi viene spesso utilizzato per controllare quantità fisiche nei processi di produzione, come nel caso degli azionamenti elettrici, che in genere richiedono l'implementazione di controlli real-time. In questo lavoro viene presentato un nuovo approccio al problema, che viene realizzato attraverso l'implementazione di un framework che affronta lo sviluppo di questi programmi tramite un approccio combinato, inteso a massimizzare il riutilizzo del codice e degli elementi attraverso diversi metodi, come "component based design" (CBC) e "construction by configuration" (C-b-C). Questo framework comprende elementi e strategie di sviluppo, sia per il firmware che per il software. Può generare automaticamente programmi con interfacce grafiche, in grado di comunicare con questi dispositivi, tramite un ambiente di sviuluppo software, che è parte di questo framework. Può inoltre generare il codice sorgente per la mappatura delle variabili dei protocolli, sempre nello stesso processo. Questo permette di mantenere coerenza con l'applicazione software che comunica con il disposivito. La parte firmware è composta da librerie progettate per ottenere portabilità e riutilizzo, limitando le dipendenze hardware solo ad alcuni moduli di questi programmi. Questo framework offre un approccio centralizzato allo sviluppo di questi programmi e dei loro meccanismi di interazione, di cui beneficiano, non solo la progettazione e la costruzione di essi, ma anche molte altre importanti attività, come la gestione delle configurazioni ed estensione delle funzionalità presenti. Si basa su un'architettura modulare in grado di integrare nuove funzionalità, utilizzando strategie e schemi appropriati, che facilitano l'integrazione di nuovi componenti. Questo approccio è stato applicato anche nella realizzazione di prodotti finiti in ambito industriale. La descrizione dettagliata di questo approccio innovativo e dei suoi meccanismi è presentata in questo lavoro di tesi di dottorato.

Abstract

The development of firmware and software elements, for the realization of electronic industrial embedded devices, involves activities which require time, efforts and many skills. The increasing complexity and evolution of the technology has brought to the need for instruments and strategies to develop and handle these programs, that are related to these products, in order to meet a reasonable time to market, reliability and efficiency. The development of these products requires different disciplines and instruments. Industrial embedded devices are often provided with softwares for their parametrization and monitoring. They usually communicate, with these instruments and other supervisor devices, through industrial communication protocols. These kind of devices are often used to control physical quantities in manufacturing processes, as the case of electric drives, which usually need to implement real-time controls. This work presents a new approach to the problem, that is done through the realization of a framework that faces the development of these different programs, in a combined approach, intended to maximize the reuse of code and elements through different methods, such as component based design (CBC) and construction by configuration (C-b-C). This framework comprehends elements and patterns, both for firmware and software. It can automatically generate softwares with GUIs able to communicate with these devices, through the use of its software editor program, which is part of the framework. It can generate the source code of the data protocols variables mapping, all in the same process, that is kept coherent with the software application. The firmware part is composed by libraries that are designed to achieve portability and reuse, limiting the hardware dependencies only to some program modules. It offer a centralized approach to the development of these programs and their interactions mechanisms, which aids, not only design and construction of them, but also many other important activities, such as configuration management and features improvements. It is based on a modular architecture that can integrate new functionalities, using proper strategies and patterns, facilitating the integration of new components. This approach has also been applied in the realization of finished industrial products. The detailed description of the innovative approach realized in this framework is presented in this PhD thesis work.

Sommario

List of Figures

List of Tables

Chapter 1

Introduction

> Programs designers, and people in general, without creativity, are like engines without fuel, they go forward only if they are pushed to.

This thesis presents the experience and the result of many years of work and study, also on the field, in embedded programming with a specific field of application. Embedded is a word that contains a world. A world made of electronic devices that are all around us and are spreading in every aspect of our lives. This huge world is often so big that it is not easy to find a proper and immediate connotation of what it includes. We can find them in domestic appliances, cars, industrial machinery, planes, phones, and many other environments.

"An embedded system is a combination of computer hardware and software and perhaps additional parts, either mechanical or electronic, designed to perform a dedicated function. A good example is the microwave oven. Almost every household has one, and tens of millions of them are used every day, but few people realize that a computer processor and software are involved in the preparation of their lunch or dinner" [7]

The results of this experience are collected inside a collection of program codes and applications in what is called a "framework". In this thesis will be discussed the meaning of that term inside the programming context and how this instrument can simplify the development of systems related to this field. Embedded devices have commons characteristics, including the technology that realizes them and some others that are specific for the application for which they are designed. The devices of interest for this work are the ones designated for the motion of industrial machinery, that are the electric drives. The increasing complexity and evolution of the technology has brought to the need of instruments and strategies to develop and handle the programs related to these products, in order to meet a reasonable time to market, reliability and efficiency. Especially for some categories of devices, the complexity and the number of disciplines involved are so many that these instruments and strategies play a necessary and irreplaceable role [8]. The framework aim to offer valuable tools to face this context. Before entering in the details of the development of the programs, software and firmware, that are involved in the functioning of embedded systems this introduction will present an overall description of this vast and important field.

1.1 Embedded Electronic Devices

Embedded electronic devices are the electronic components designed to fit a specific context and needs. The reason behind their diffusion is that these kinds of devices are able to monitor and control a large variety of systems, without the need of human supervision. The range of applications needs and constraints, brings to a significant variance between devices. There are also a lot of international standards that devices need to fulfill, in order to be used in specific sectors that define some device classes with a certain degree of flexibility. The majority of the embedded devices produced nowadays are characterized by the presence of digital processing by the mean of a Core Processing Unit (CPU) or programmable hardware logics, like Field Programmable Gate Array (FPGA) or both of them. The presence of digital processing implies the presence of programs and codes, which are the sequences of instructions executed by these devices in order to accomplish tasks accordingly to

a process logic. When the digital elaboration has a consistent part in the project, the related design complexity and usage, requires significant efforts, but gives significant advantages. Software and Firmware engineering are the disciplines behind the design of programs and application logics that develop methods and strategies to handle the complexity of translating the physical word processing into an equivalent digital image with which the electronic devices are able to interact. Depending on the application, devices can have different computational power that usually depends on the needs. Embedded electronic devices and electronic devices in general, are made of three components:

- Hardware: is the physical part of the device and comprehend the digital elaboration core and the various electrical circuits deputed to signal and power processing.
- Firmware: is the program directly executed by the core unit at the lower level of abstraction that is directly connected with the hardware elaboration capabilities of the core.
- Software: this term usually identifies the program code that is built on the firmware of the device at a higher level of abstraction and unlike firmware is less dependent by the core platform. It usually relies on operating systems (like MS Windows or Unix), that are programs that create a homogeneous substrate for code instructions that are able to be executed in similar way from different elaboration cores, while firmware instructions depend on the instruction set available on the specific elaboration unit. Software usually is not executed by the device, but from a general-purpose computer and interacts with devices through communication channels.

It is important, even for an expert of the sector, to remember these definitions, because often digital products are all labeled with the term Software, but due to the different context and usage of these programs they need to be treated as separate elements with a proper handling. In embedded programming it is easy to encounter operating systems that are slightly different from the computer's ones (for example

MQX-RTOS from NXP [9]). The main difference between them is that embedded operating systems are designed for some specific CPUs and offer a substrate that is very different from the computer one especially for the way in which programs are written. This depends on the different abstraction in which these programs operate and consequently the paradigms that have been developed in these disciplines. Software programs operate inside an operating system which offers a virtual substrate that handle resources like memory, CPU elaboration and others, totally masking the hardware capabilities of the machine. This has allowed the development of particular programming techniques, the object-oriented programming (OOP), which have been designed to maximize the reuse of code and drastically reduce development time. Firmware programs, due to their specialization, cannot benefit from similar instruments. The reason is behind the different scopes of the two elements, Software has been meant to be more general purpose with the aim of develop more with less effort, paying the price of a loss of performances due to the resource consumption of the substrate, which is the operating system. Firmware has been intended to interface directly with the execution platform, so it can be optimized and take advantage of the whole available resources, but the development effort is higher due to the complexity of handling all the hardware related features, including also the time spent writing the relative code (which is already done in case of OSs). The simplest devices are made only of hardware that processes electrical input signals and produces electrical output signals. Adding the digital processing with firmware and software elements increases the complexity of the device, enhancing its capabilities, but this enhancement has also to be supported by the proper hardware, that is one the reason that add complexity to the design of devices with significant presence of digital logic. The case in which hardware, firmware and software elements are present in the design of devices is the actual trend, that is one the reasons behind the multidisciplinary nature that is emerging in this field. In order to handle the related complexity, the disciplines related to the "virtual" components of the system have developed approaches to the engineering of these products. The collection of these instruments and methods is often referred to as a "Framework", which in this field, identifies the overall structure of the project that has the aim to provide a predefined structure for the programs

design in order to facilitate the whole process. Frameworks can have different formalizations and purposes and can be supported by other instruments. For this kind of devices, it is becoming quite common to have boot-loaders, interfaces for signals, additional memory storage, communication protocols and in some cases embedded RT-OSs (real time operative system). It is not difficult to believe that the firmware that is needed to handle these devices grows in complexity as the device itself and consequently how an instrument capable of supporting this process can be useful.

1.1.0.1 Hardware

The hardware that composes electronic embedded devices is strongly "vision driven", that means it is related to the environment in which it is applied, so it is its design and the electronics components that realize the device. In this work, the discussion concerning hardware will not interest the design aspects of it and will only analyze the relevant aspects in the building of the program logic that controls the device. These embedded devices, as said before, are made to monitor and control a variety of systems, and their functioning can be described with the classical paradigm sense-plan-react. The existing available technology for these kinds of embedded devices includes sensors, active and passive components, and electronics components designed to interact with physical quantities specific for fields of interest.

Sensor and actuators convert physical quantities in electrical signal and vice versa, the device logic interpret these signals and plan the corresponding actuation that depends on the logic. From the computational point of view, the device logic, can be realized with different technologies, in this writing the attention will be focused on the design for devices with one or more CPUs, that require the engineering of a program code for their correct functioning. The "plan" stage nowadays has made a significant step forward and can include very sophisticated logics and even artificial intelligence algorithms. The currently available technologies provide a large portfolio of micro-controllers and digital signal processors (DSPs) and other elaborators which are for example FPGA or other reconfigurable digital hardware capable also of em-

bodying a central processing unit. The central processing unit executes the program logic in which algorithms and complex logics are coded. There are devices which do not have a CPU, without program logic or which are able to perform their tasks with the hardware only, but generally at least a small elaboration unit is always present in modern devices.

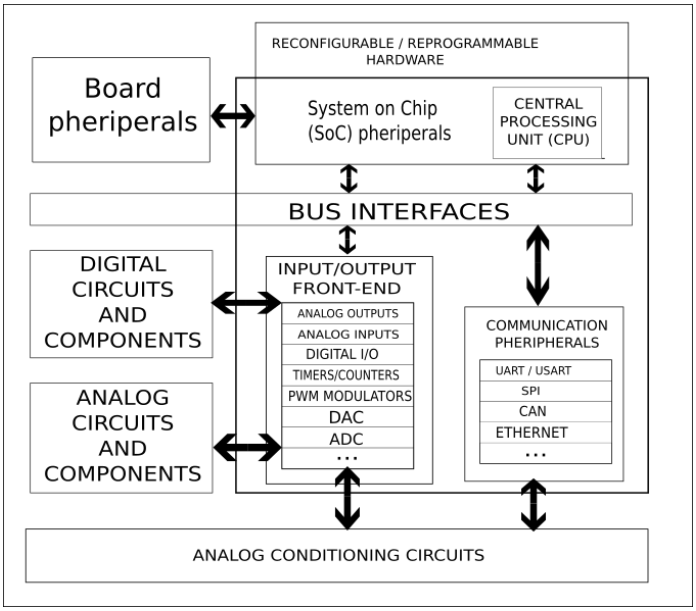


Figure 1.1: Generic hardware schematisation of contemporary devices.

Modern devices tend to have a lot of features and hardware components, see fig. 1.1, that elaborate analog and digital signals. The CPU, is able to interpret only digital values, so there are dedicated hardware circuits, like analog digital converters (ADC) and digital analog converters (DAC), which are used as interface with analog circuits and can convert electrical signals into digital values and vice versa. There are a lot of dedicated circuits which are designated to interface with a variety of electronic components and to perform many tasks such as communication with other devices, memories, converters and so on, which are called peripherals.

The most widespread CPU building technologies that can be found on a board

are:

- System on chip (SoC): SoCs are a collection of circuit built around a core unit which provide a complete set of peripheral circuits suited for certain applications, which diffusion make possible to design components that are specific for the need. These systems are convenient because they minimize the circuitry needed to drive the external circuits on the board and connection buses between internal components are optimized in the design, but their flexibility is limited with respect to other solutions and it is vey important to choose the SoC that have all the necessary items for the application.

- Standalone CPU: Some controllers are provided with the minimum set of peripherals, the essential for programming and interconnection with external circuitry, the design of the system is more flexible but requires more effort in the design of the board.

- Configured CPU: Configured CPUs are more a possibility that a real case, except for prototyping purposes, with FPGAs and other reconfigurable hardware elaboration unit can be configured and interfaced with the digital array which can compose many and different digital circuits. Often these instruments are coupled with external CPUs in order to have a built-in core for elaboration surrounded by a more flexible hardware.

From the programs design point of view, the technology that realize the hardware influence only some part of the firmware, or at least it should. This is one of the themes that will be explored in this writing and how the impact of this influence can be handled and what could happen if not.

1.1.0.2 Firmware

The Firmware is the program code directly connected to the hardware, of which in some part is dependent. It has to perform the handling of the peripherals connected to the processing unit through registers and buses which can exchange digital information between circuits and this information can modify the electrical behavior of the

device in order to accomplish the sensing and actuation needed to interface with the physical world. It also has to execute algorithms designed to mathematically and digitally represent the systems they are controlling in order to follow its evolution and control their behaviors. The different purposes of these code portions and files define the abstraction in which they are identified. The hardware dependencies and the many possibilities that are in present in the algorithms construction makes this discipline quite challenging. The Firmware also needs to be reliable because it is always provided with the device and it is responsible of its correct functioning. Modifying it can be dangerous, especially for environments in which certifications of security are needed and any modifications implies the repeating of the whole testing process. Embedded devices are expected to run continuously for years without errors, and in some cases, recover by themselves if an error occurs.

1.1.0.3 Software

Software is a more commonly known term, even for people who are not experts of information technology disciplines. Everyone that uses a personal computer has used some kind of software, or at least knows that is a “computer thing”. They are also using Firmware but unlike software, they would not notice anything because it usually does not have what is called “presentation”. Presentation is a term that identifies the program user interfaces, graphical, textual or of other kind, with which the user can interact to access software services. Firmware perform its work in a “silent” way, that means the user often does not have any feedback of what is doing, while software has quite always the opposite manner. Software programs are assembled and executed inside operative systems and can be used and constructed on every device capable of hosting the proper operative system. They are made to operate in a more general-purpose environment in which computing constraints can be relaxed, exchanging the performances with adaptability. In the embedded context, software covers two different parts of the design, one is related to computer aided design (CAD) programs, which guide design and development of device boards, firmware and software components. The others are the software instruments developed to handle and interact with the device. This last class of programs cover many sector of the electronic

world; programmable logic controller (PLC) and human machine interfaces (HMI) and industrial PCs, which are used in industrial environments; Mobile applications for phones and tablets and PC application for general purpose computers. PLC and HMI are particular embedded devices that have have their custom programming environment to implement some automation logics, they are some sort of trade-off between an embedded controller and a general purpose one. Industrial and common PCs can use identical operative systems, with the only difference that industrial PCs are able to operate in harsh ambient. The major difference in Software programming applied is the real-time capabilities of the operative system and the relative design of the applications. Software programs, differently from Firmware, do not need to be aware of the hardware that will execute them and the development is purely related the program instruments and language used. Another big difference is that Software makes extensive use of graphical user interface (GUI), which has a significant impact on the user experience and the ease of use. There are significant differences between the hardware (the "physical") and the program ("the virtual") elements, but also between the virtual elements which brought to the term hard/firm/soft-ware. With the technological evolution, the possibilities made available by the virtual instruments are becoming furthermore important and consequently their development. Proper development strategies can make available these possibilities and many other strategical advantages.

1.2 Embedded Systems

Embedded systems can be composed of different embedded devices, especially with new emerging trends and improvements, like internet of things (IoT) [10], in which more embedded devices are placed together to realize structure capable of doing more activities and works. Many systems are realized with devices that are executor units (known as slaves) and supervisor units (known as masters), to avoid term that personally I dislike, in this writing, slave units will be referred as executors and the master units as supervisors. Executor units communicate usually with one supervisor unit that coordinate them through a communication bus that links all them together.

The way in which they communicate and the supervisor activity are often defined in affirmed protocols, especially for industrial devices and internet connected systems. Devices need to implement the proper communication in order to be of relevant use in the realization of modern machinery.

"Embedded systems engineering deals with a number of highly complex challenges, which, until now, have not been sufficiently addressed by process support. Specifically, there is a need for integrating multi-disciplinary, multi-lifecycle, multi-site, and multi-organization approaches, due to domain-specific characteristics." [11]

1.3 Embedded Programming

Embedded programming involves different activities, once the hardware specifications are gathered and it is clear what are the purposes and what a device has to do, the proper digital controller or controllers are chosen in order to have enough hardware peripherals and computational power. The digital controller (or controllers) choice implies also the programming instruments and environment to compile the program, transfer it into the microcontroller memory and perform "debug" activities (which means removing errors from the program) which are core part of the development. The device programming usually follows some steps:

- Setup of the programming environment.
- Prepare the program structure which can include software development kits (SDKs) which are program libraries of preassembled functions to use microcontroller peripherals and capabilities with some use case examples in order to make the set up faster.
- Write preliminary code to test hardware functions and capabilities to check eventual need of hardware revision.
- Every piece of the written program should be tested and documented.

- Compile the program and iterate tests until the use case scenarios are satisfied and the system is considered reliable.

This kind of work requires a lot of time and resources that is the reason behind the need of preassembled code and instruments in order to reduce development time. The use of third-party libraries and preassembled piece of programs bring of course a lot of benefits in addition to the time saving, which can be the use of tested algorithms which functioning has been already proven, but it also need a program structure capable of integrating all these instruments together.

"In the last decade, the concept of modularity has caught the attention of engineers, management researchers and corporate strategists in a number of industries. When a product or process is "modularized," the elements of its design are split up and assigned to modules according to a formal architecture or plan." [12]

The reuse of programs code can be done also by the same developers which can use some strategies to develop portable code and reuse it in more projects and devices. In these methodologies often appear concepts like component based development (CBD) [13] and component based software engineering (CBSE) [14].

"The widespread use of embedded systems mandates the development of industrial software design methods, i.e. computer-aided design and engineering of embedded applications using formal models (frameworks) and repositories of prefabricated components, much in the same way as in other mature areas of engineering such as mechanical engineering and electronics." [15]

The need of techniques, knowledge and methodologies to handle this technological evolution is something that started more than twenty years ago but is still a big challenging issue, finding strategies which can be modular, flexible and able to adapt to different constraints such as real-time applications and different platforms, is something that need a difficult refinement process in which the trade-offs cannot be avoided and a general solution often would imply an unacceptable loss of perfor-

mances and resources with a significant cost increase to realize the project.

1.4 Devices Categories

As already introduced, device's fields of application are countless and inside every field the variety of devices is equally vast. Categories of devices are sometimes defined in international standards, which can be about the whole device or only parts of it. These standards are prepared by international organization, like International Standards Organization (ISO), International Electrotechnical Commission (IEC) and Institute of Electrical and Electronics Engineers (IEEE), which have the role of giving guidelines in order to have compatibility and reliability of the products all over the world. The fitting of a particular device to a category can regard different level of abstraction and the role that is supposed to perform. It is easy for some devices to overlap in many associations. International standards are more focused on functional parts of devices, process handling and the regulation of the involved sectors, especially for safety related aspects. For example, electrical drives are a well-known category of embedded devices whose guidelines are defined in international standards but also from international consortium of companies like CAN in Automation (CiA), which develops standards for one of the most used protocol in the industrial sector and also has defined other guidelines for machinery which has been taken as reference by other consortia and organizations. Standards adoption, especially consortium's ones, has always generated controversies, because people often felt that leading companies were forcing the whole world to follow their rules, which can be true under certain points of view, because companies pursue profits and interests, but from another point of view they are necessary, because if something has a wide usage, maybe could be because it works well and if one wants to do something similar, at least it should be able to cooperate with the existing world. Device categories can be formulated taking into account functional aspects and features, for example the IoT is a particular category which has the main purpose of interfacing embedded systems with the Internet. Inside this category there are wireless devices, with wireless communication media and protocols, and wired ones. They can be used for

example in domestic or industrial environments. The combination of these features defines a collection of sets which can identify device classes from a functional point of view. Electric drives, for examples, can control different kind of motors, can be commanded only by physical signals or with high-level communication protocols. Depending on the complexity of the application the features that they have they can be very different, generating a collection of subsets even inside the same class. There can be several categories, sensors, actuators, presence of communication protocols, the presence of floating-point arithmetic unit (FPU) inside the core elaboration unit, fields of application, constraints, regulations and many other characteristics. This is the main reason behind the interest in modular design and CDB, that means having functional blocks of hardware and code which can be reused in common classes and intersections of them. Concerning the reuse of the code elements, there is a big difference from software to firmware elements, because platform specific code even realized in a modular and portable way it can be used only on the same processors and also some algorithms can be optimized for the specific processing architecture, such as the ones containing a FPU, which permits the hardware computation with floating point numbers. The execution of the same algorithm on a CPU without the FPU will quite always result in an excessive amount of time for quite every application.

1.5 Electric Drives

Electric drives are a particular category of embedded devices which is deputed to the electric motors motion and machine control, their usage in industry is increasing especially with the diffusion of brushless motors which have a high torque and power density allowing their application in many motion solutions. The capabilities of these devices of executing high precision motions is one of the other important aspect regarding their diffusion. There are many topologies of devices depending on the type of motors and sensors they can handle, the operating voltages and currents involved that determine the power range of the drive. The peculiarity of the electric drives is that they can be driven by digital and analog input signals or through communication buses supporting high level protocols to control the physical quantities that are in-

teresting for applications such as speed, torque, position and acceleration. Often the drive has to implement some logic to handle mechanical factors which are involved in the motion transmission. All these features require the knowledge of different fields and the ability of translating it into programs, a specific chapter for these devices will be treated in this work showing the strategies adopted to solve real world applications scenarios, presenting the specific characteristics of this device category.

1.6 Generic Functioning of a Contemporary Embedded Device

Despite the realization technology, reconfigurable hardware, FPGA or Digital Signal Processors (DSPs) or a SoC system which has both, a general execution behavior of this kind of device can be described:

- When powered, devices, execute the boot phase, which consists in loading the program machine code from a source of information, which can be a memory or a communication channel.
- Then memory and registers are initialized to a default condition (the start-up values), specific for the elaborator, and finally the first program instruction, which is the “entry-point” of the user program, is loaded.
- After that, peripherals and the internal hardware are properly configured for the application.
- When the initial configuration has took place, it is usual to load stored parameters and the digital image state from a non-volatile memory which can be internal or an external one with respect to the core unit. The way in which this stage is performed depends on the storage capabilities of the device.
- With the new state parameters it is sometimes necessary to reconfigure the peripherals and the other hardware components.

- After these stages, the main routine and the configured subroutines and interrupt request routines (ISRs) start to execute application tasks.

All these aspects will be expanded and treated properly in the chapter about the firmware development. Besides the main execution routines, there are also the ISRs which are handled by one of the most important modules, that is the interrupt controller which can trigger the execution of periodic and asynchronous execution flows, always supported by priority mechanism and basic scheduling principles. The way in which program execution flows and routines are managed depends on the hardware that performs the elaboration, and implementation details are often related to that. This work will be focused on programming SoCs which have a CPU, because these are the platforms on which experience and use cases have been matured.

1.7 Firmware and Software Engineering

Software engineering is a term that has been heard more than Firmware engineering and they often are considered both as Software. The day that the most of the people involved in information technology and electronic environments will use these two terms separately will probably be the day in which we will have a significant improvement in the related products. Why should these two definitions be kept firmly separated? Well, in the previous paragraph the description of these two elements should be enough to understand that they are quite different things even if it is right to say that it is always a matter of writing program code. If this is not enough to justify the above sentence, the following paragraphs and chapters will show how these differences are significant and how reducing these concepts as only one will result in a severe loss of functionalities and strategy oriented development. There are common elements between these two disciplines, similar techniques of algorithms implementation can be used in both, also some programming languages can be used both for micro-controllers programming and software implementation. For example, the “C” languages and “GNU C” compilers can be used with the same instructions on PCs and to build firmware applications, of course with differences in the performances, in the context of application and compilers targets. Taking as a reference international

standards, the design process of a program can be covered with the themes listed below, which are the subject of the chapters of the Software engineering body of knowledge (SWEBOK)[16] from IEEE Computer Society (one of the most important organizations in the world for information technology). Every chapter sums up the legacy of many experts in the field which have worked in the development of the information technology systems that are used today.

1. Software requirements: the process to understand and document the specific characteristics that a program has to satisfy.
2. Software design: the process of modeling and determining the architecture, logics and elements of the program.
3. Software construction: the effective implementation of the algorithms and logics that are needed to build the software.
4. Software testing: the process in which use cases are proven to be reliable, according to defined procedures, and the software "bugs" are removed.
5. Software maintenance: the process that follows the software during its lifecycle and grants the normal operating condition.
6. Software configuration management: the process of handling all the versioning and upgrade or customization of the product.
7. Software engineering management: the management of the various aspect of the design of the program.
8. Software engineering process: the supervision of the whole process.
9. Software engineering models and methods: the instruments which are used to build programs.
10. Software quality: the process that aim to grant a development which is able to satisfy general criteria which usually grant the achievement of a reliable product.

11. Software engineering professional practice, involves many aspects, also ethical, related to the evolution of the software development.

12. Software engineering economics: the process which should follow the commercial aspects of the software products.

Knowing what these points are about and understanding the processes they rule is a good way to avoid problems and commit mistakes which have been already encountered and on which someone has put a warning sign. That does not mean that one who does not follow exactly these guidelines will necessarily encounter problems; everyone can have their own strategy but if one is not familiar with the concepts of these points, they are more likely to encounter a lot of problems in the development of a useful software [17]. Often the development of a program does not follow all the provided guidelines, sometimes other strategies are preferable, like Agile or Rapid development which are forms of development which need less time, resources and experience in order to produce a complete software product. But generally, the structured approach lead to better result and approaching complex program without it, could result in an unmanageable project. These are consolidated practices which are intended by experts of the discipline and have been followed in the past years due to the benefits that they introduce. Having an ordered approach to the handling of the program complexity, which is quite always a big challenge, can make the difference. These practices which were affirmed in the software development are often used also in the firmware, but their application has often generated some misguidances.

"There exist various general strategies to help guide the design process. In contrast with general strategies, methods are more specific in that they generally provide a set of notations to be used with the method, a description of the process to be used when following the method, and a set of guidelines for using the method. Such methods are useful as a common framework for teams of software engineers." [16]

As stated by the above sentence, from the IEEE SWEBOK, the general guidelines are useful to define the infrastructure and the guidelines to orient ourselves inside a

project, but what is referred as methods, strongly depends on the specific instruments that are used to develop the project and a general strategy should also consider them in the overall plan. Even if there are some common elements, software engineering has evolved in a discipline which is specific for the development of programs inside high-level OS. This kind of development is almost always driven by object oriented languages and paradigms. It will be discussed in detail in the chapter about the software development what are the characteristics of these methodologies, and will be explained the differences from the ones used in firmware programming. The development of a program need a set of instruments that are not only the programming languages and compilers to build the execution code, but there are also Integrated Development Environments (IDEs) which integrate a lot of instruments to support the entire process. The development is often sided by the modeling of the application components and behaviors through appropriate modelling languages, which are a set of symbols, notations and graphs which are studied to express the interested features and characteristics of a program in easily understandable schemes, like a sort of navigation maps to the program. Making a comparison with a building, these schemes are like the planimetry of a construction. The unified modeling language (UML) [18] is one of the most used and affirmed in environment which is made for object-oriented paradigms. During the last decades, Software development has created and consolidated specific methods and tools in order to handle the complexity of the incredible growth that programs have faced. The complexity of firmware programs was, in past years, more related to the implementation of digital algorithms and models rather than the growth of elements and components, but now they are facing a significant progress and a change in paradigms. Devices have now the possibility to have a significant computational power to implement complex logics and services, and they are becoming actors inside a network rather than only components. What is missing now, or at least what many feel missing in the firmware development is the consolidation of practices and methods proper for their field of application. There are a lot of interesting works that are proposing solutions and innovations, and this work hopefully will do the same. There are some attempts also on using methods that were involved in the software also in the firmware, as the usage of UML modelling also for lan-

guages that are not object-oriented. It can be done in some cases but, for example, using class diagrams to model a program in which the language does not use classes could be confusing. My personal opinion about that is is reflected in the sentence took from the SWEBOK, methods should be specific for the context and take into account the elements and instrument involved, then strategies should be able to give general guidelines for framework and architectures in which they are applied, covering all the needed aspects, from design and construction to testing, documenting and maintenance of the code [19], [20], [21], [22].

1.8 Frameworks

What are frameworks? A definition which I found appropriate is the title of a paper, which is **"Frameworks = (Components+Patterns)"** [23].

An appropriate definition in few words, that sums up the main concepts of the frameworks, which are the combination of many components organized with common schemes and methods, which are the patterns. More specific definitions can be found, in this case regarding software frameworks, due to the reference to the object-oriented programming, that gives an idea of what a framework is.

"An appropriate combination of object-oriented programming concepts allows not only the development of single reusable components but also of semi-finished architectures (= frameworks)" [24] .

A framework is a collection of guidelines, structures, and every other element which is useful to organize a collection of components subjected to specific constraints and operating manners which need proper methods and rules for they correct functioning [20]. Frameworks can be used and specialized for every aspect that concerns software and firmware development, design, testing, modeling and everything that is included in the specificity of the environment in which they are applied [25],[26],[27], [28]. They should be the base on which teams, organizations or single developers bases their present and future works. There is not always the need of a framework to develop a digital product, but in case of products which need long time of development and a long lifecycle it is something that at least has to be taken into

consideration. It is also extremely useful when there is the need to extend or develop new products.

1.9 Framework for Embedded Electronic Devices

In the embedded electronic world, the term framework, is often used to identify all the virtual elements necessary to build the logic of a programmable device. These elements are program code, libraries, software instruments which aid the designers to build application logic related to their devices. Frameworks are not only involved in the construction of a program, but as already introduced, they can cover different activities inside projects [29], [30], [31], [32], [33], the ones who aid the design and construction are of course extremely useful because a lot of time and efforts are spent in this stage [34]. Without these instruments, the design process will take an amount of time that will quite always make the project unfeasible. Many of them are used in testing and model validation [35],[36], [37],[38],[39],[40],[41]. A framework for embedded electronic devices usually covers a defined class of devices in which it helps to handle the specific characteristics of the class [42], [43].

In this thesis a framework that helps the development of software application and firmware for electric drives will be presented, underling the benefits of some specific methods in the development for what concerns reducing time of development, reuse of code, flexibility and reliability of the programs.

1.10 International Standards and Information Technology

In the embedded electronic field most of the relevant standard are produced by the International electrotechnical Commission (IEC) such as: Security in information technology (IT) systems ISO/IEC 27001; Floating point arithmetic for microprocessors ISO/IEC/IEEE 60559; Medical standards for electronic devices IEC 60601; Adjustable speed electrical power drive systems IEC 61800; RFID: ISO 18000, 15961, 15962, 15963; and many others. One of the most important standards is the 61508 which provides guidelines for the safety in many industrial sectors, which is relevant

for certifications. ISO 26262 is an adaptation of IEC 61508 for Automotive Electric/Electronic Systems which is adopted by most of the car manufacturers. Before the launch of ISO 26262, the standards for safety related automotive systems mainly consisted in by Motor Industry Software Reliability Association (MISRA) guidelines. The MISRA project was conceived to develop guidelines for automotive embedded software and firmware in road vehicle electronic systems. A set of guidelines was published in November 1994 [44]. This document provided the first automotive industry interpretation of the IEC 61508 standard. Today MISRA is known mainly for the guidelines regarding the use of C and C++ languages for compliance certification. MISRA-C is the de facto one of the most adopted standard for embedded C programming in the automotive safety-related programs components. IEC 62279 is a specific interpretation of IEC 61508 for railway applications that covers the development of software. The process industry sector includes many types of manufacturing processes, IEC 61511 is a technical standard which sets out practices in the engineering of systems that ensure the safety of an industrial process. IEC 61513 provides requirements and recommendations for the instrumentation and control for systems safety inherent to nuclear power plants. IEC 62061 is the machinery-specific standard regarding IEC 61508. It provides requirements that are applicable to the system level design of all types of machinery safety-related electrical control systems. International Standards Organization (ISO), IEC, IEEE) and MISRA are not the only organizations involved in the standardization processes, there are international, continental and national organization which cooperate for the diffusion of practices and guidelines granting their correct application from global to local. Other organizations which have the merits to be cited are the internet engineering task force (IETF), World Wide Web Consortium (W3C) and many others which are behind the development of Internet protocols, standards, like the widely everyday used Hyper Text Transfer Protocol (HTTP) in RFC 2616 and RFC 7540, and the evolution of the Internet. When working with the development of programs which will be in devices that are used all around the world and that have to coexist with the already existing universe it is important to follow common guidelines and that is the reason why every engineer should be aware of the organizations of colleagues that are in charge of this process.

There are many controversies due the fact that often these standards are available through the payment of fees, but when there is the intention of producing commercial products there is also the possibility to cooperate and receive support by these organizations. Anyway, there is often the way to access these technical documents trough drafts that are released freely by the same organizations; of course fees and affiliations, for commercial and industrial products, are also a way to contribute and have an assurance of receiving and granting a service. Consulting the international organization documents can give a great contribution to the technical knowledge of a programmer on these subjects, and before approaching a problem a research on these documents is suggested.

1.11 Embedded Devices Software and Firmware Engineering

The electronic embedded devices, deputed to the control of systems or parts of them, especially in the industrial environment, are often used as nodes of a bigger system. Industrial devices such as electric drives, communicate with other devices and supervisor units, through communication protocols. They are almost always provided with commissioning software, especially in the case of electric drives [45],[46]. These softwares are used to configure their setup and functioning. Softwares can connect to these devices through the proper adapter that acts as interface for the physical channels. These software applications have complex and rich graphical user interfaces (GUIs), through which devices can be parametrized, monitored and controlled. Devices can also interface with other supervisor units, always through communication protocols, like PLCs and other controllers, and not only their own commissioning software.

These two elements, in order to be able to establish a communication, have to define a common data map, which is the variables addressing. Every variable that has to be shared in the communication process, has to be properly placed inside tables or dictionaries accordingly to the protocol specifications.

The firmware and software development is usually based on the usage of specific

integrated development environments (IDEs) for these two kind of programs. IDEs integrate software development kits (SDKs) and other third party code libraries to simplify the writing of the program code [47].

These realization of the device programs, commissioning ones, and CAD softwares require many different engineering and coding activities.

- Construction and engineering of the firmware program.
  - Integration of SDKs and third-party libraries.
  - Development of specific algorithms for the control of the specific system inherent to the field of application.
  - Development of the code related to peripherals and hardware circuits handling.
  - Implementation or integration of communication protocols code. Even when libraries provide a complete protocol logic, there is often the need of mapping the device variables in a proper dictionary. The need to build exchange mechanisms and integrate them with the lower layer of abstraction of the code program.
- Construction and engineering of the software program.
  - Development of the application logic.
  - Implementation or integration of communication protocols and adapters APIs.
  - Design and construction of the GUI.
  - Design of device variables access mechanisms, using protocols functions, according to the device dictionary.
- Design and construction of the interactions between supervisor application and device.

The enumerated activities involve the knowledge of different subjects and disciplines. These are difficult and complex engineering tasks, which require efforts, time, expertise and proper approaches.

In order to reduce the development time of these products, the reuse of code is essential and also recognized by the software engineering discipline. SDKs, libraries, design strategies and other tools are intended to obtain such desired characteristic. Reuse of code and methods, it is not only a way to reduce development time, but also a way to increase reliability of the systems, through the use of code that has already been tested. Programs are products in constant evolution and the development process itself is repeated until it is proven that desired behavior is achieved. So programs evolve through versions of themselves, in which coding activities and revisions make some modifications between a version and another. Adding a single line of code can drastically change the execution flow of a portion of code and have impact even on the whole program, so it is easy to compromise the result even with a single apparent harmless instruction. The reuse of code with a known sequence and execution can also decrease the probability of facing this scenario. Design strategies that can increase the reuse of code can also aid the development of modular structures in which subdivision of code, through libraries, files and functions, can aid to isolate some aspects of the behaviour in some specific part of the program which helps the understanding of it and when there are unexpected flows of execution is easier to locate the wrong instructions sequence. Understanding a program is essential to maintain it during its life-cycle and being able to use it to produce further development, which is also the reason of the usage and need of modelling languages, which are an indispensable aid, especially for complex and huge in size programs. In the following chapters will be presented some methods and strategies that can enhance these and other desired features that aid these processes.

Chapter 2

Industrial Embedded Devices Framework

"You might know Daedalus as the mythological prisoner who fashioned wings of feathers and wax to escape from the island of Crete with his son Icarus. But it was as architect and sculptor, one said to have designed a labyrinth for King Minos on Crete, that he earned his name. Daedalus (from Greek daidalos) is Latin for "skillfully wrought." The same skillful Latin adjective gave English the adjectives daedal (in use since the 16th century) and Daedalian (or Daedalean), a synonym of daedal."
(From Merriam-Webster online dictionary, paragraph "Daedal and Greek Myth")[48]

This chapter present some concepts behind the realization of the D.A.E.D.A.L framework (Development Aid for Embedded Devices Automatic Labor). This framework collects another software one, that will be introduced in the next section, which is part of the entire system, firmware code libraries, methods, patterns and techniques to aid the development of the firmware and software elements of embedded devices, especially for the most repetitive and laborious tasks.

2.1 Improvement Proposal

Section 1.11 sums up the main activities related to the engineering of the program components of electronic embedded devices and the complexity of this field, which has also been in part depicted in the introduction. As will be discussed inside the corresponding chapters, software and firmware design and construction require a significant number of time consuming and complex coding activities which impacts on the entire life-cycle of the final products; testing, maintenance, version and configuration management. In some collaborations with local industries that operate with industrial embedded devices and especially in the electric drives field, see appendix B, and inside the academy, I have faced the development of many firmware and software programs parts and even frameworks (section 1.8), see fig. 2.1 and fig. 2.2 in the next section.

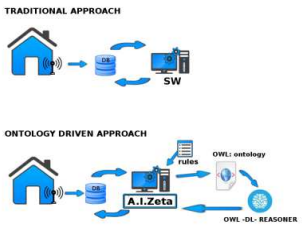


Figure 2.1: A.I.Zeta framework [1].

Fig. 2.1 represent a developed framework based on artificial intelligence knowledge bases with inference mechanisms, that has been developed for ambient assisted living (AAL) scenarios [1].

From the acquired experiences and knowledge about embedded development, it has been noticed that many tasks could be done with some automatism or with some instruments that could help the development and aid to handle the complexity of this field. The idea that slowly growed was the development of a framework which could include the firmware and software development aspects and their strict interaction. D.A.E.D.A.L, like any other framework, is composed by components and patterns,

which are intended to give an infrastructure to approach specific context problems through a structured method. These elements treat software and firmware development and their interaction, offering a centralized approach to the programs development in the industrial embedded devices context. The elements inherent to software development are handled by an internal software framework which also handles some interactions between software and firmware components. It mainly relies on two software applications which can compose and generate software programs with rich GUI and communication features, through a graphical editor, without writing code. This software is also able to generate the corresponding device dictionary files and keep them synchronized with the generated software interface with which it shares data. The software program acts as a commissioning one and can monitor, control and parametrize the functioning of devices. For what concerns Firmware modules, the adoption of particular design patterns and applying them in the writing of code libraries is possible not only to reuse them as it usually is supposed to happen with them, but also to have a modular structure in which elements can be more easily added with an effective component based design fashion. How these features are realized will be explained in chapter 3 for the software part and in chapter 4, for the firmware ones, after the introduction of the overall system and main concepts. that is presented in this chapter.

2.2 Architecture Description

The architecture of this framework, that is presented in this chapter, has been studied after a first attempt to realize a small software framework, see fig. 2.2, for the construction of a PC-program with graphical user interface to monitor, control and parametrize the functioning of a family of electric drives, made by Temec Drive, a small company that operate in the sector of electric drives design, see annex B. AZ2 devices are electrical drives for dual DC motors with some specific implementation for linear actuator based systems which use Modbus protocol to communicate with other devices and supervisor elements. The developed software, named AZ2-Temec-Interface is actually used and distributed to handle AZ2 device family.

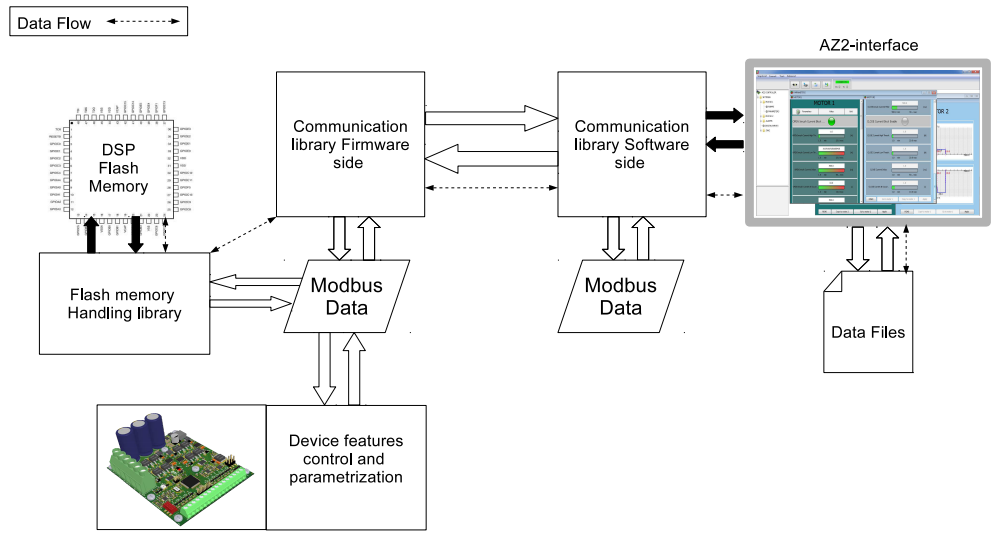


Figure 2.2: AZ2 Framework.

The design is data structure centered, both for the interface generation and functional features. This was the first attempt to automatically generate the user interface and the data-handling. Through a software instrument is possible to define the data structure with which the device could interface the features of the drive, and the same structure is also used to generate the user interface components without the need of writing any line of code. This System was able to adapt rapidly to the evolution of the device features, but was extremely specific for the AZ2 Drive and with very limited components, but at least for that project has contributed to a very flexible and modular approach. The important concepts that have been recovered and revised are:

- The automatic generation of software components through data.
- The use of data as neutral interface between software and firmware modules.
- The building of a software instrument which generates data structures though a guided approach.
- The application can be extended without writing code, and the introduction of new elements which require coding is done though the extension of the

software configurator and inserted in the application only when the new module is tested and the corresponding data equivalent is released into the application libraries.

The last point was not totally achieved in the realization of this project because there were a lot of customizations inside the software interface so it was not possible to extend this implementation to other devices; this is the first big improvement of the realized framework in which there is the possibility of building interfaces for more devices without writing additional code.

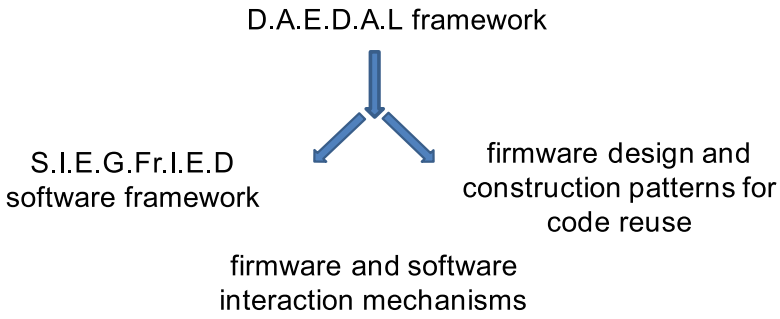


Figure 2.3: D.A.E.D.A.L framework concept and composition.

The framework D.A.E.D.A.L consists, as already said, in software and firmware programs and elements. The software part comprehends the two modular applications which contain libraries to handle communication, data and devices specific features, this part is the Software Interface Environment Generation Framework for Integration of Embedded Devices (S.I.E.G.Fr.I.E.D). Software Interface Environment Generator (S.I.E.G) application is the one deputed to the automatic generation of the software interface program, which interacts with the device. The other one, the Configuration Integration Tool (C.I.T) is the the program in charge of the construction of the features of the application, both for functional and graphical elements, that are represented by a proper data structure, that is interpreted by the interface generator. Their

design and functioning will be further explained in the next chapter. S.I.E.G.Fr.I.E.D is the acronym for the software framework, containing S.I.E.G and C.I.T applications, that compose the D.A.E.D.A.L framework. The realization of this system follow different strategies for the overall functioning, one of the fundamental one is the Data Structure-Centered Design, see 3.1.1.2, in which there are data and elements that frequently access and follow them. The firmware elements are developed to interface with data structures which are the bridge between software and firmware. At the moment these libraries provide a set of functionalities mainly focused on the elaborators that have an FPU unit. Data structures are also used by the software interface and the integration programs as link. In this last case there is behavior which is not proper of the used architectural strategy, that is the generation of the data structure even if its generation could be considered as an access to it.

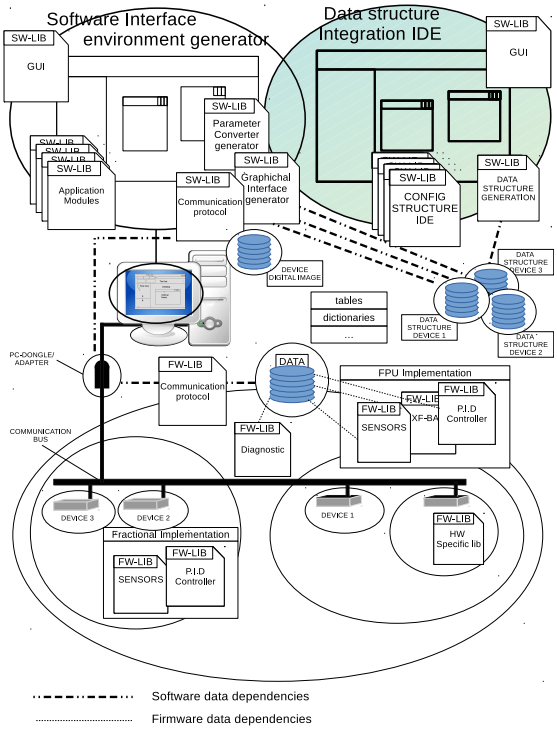


Figure 2.4: D.A.E.D.A.L architecture and elements.

2.2.1 Data Structure-Centered Design

This strategy implies the presence of two elements:

- A central data structure or data store or data repository, which is responsible for providing permanent data storage. It represents the current state.
- A data accessor or a collection of independent components that operate on the central data store, perform computations, and might put back the results.

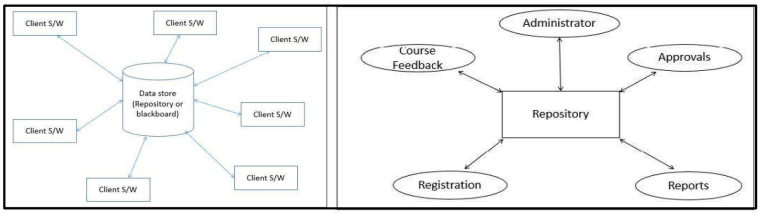
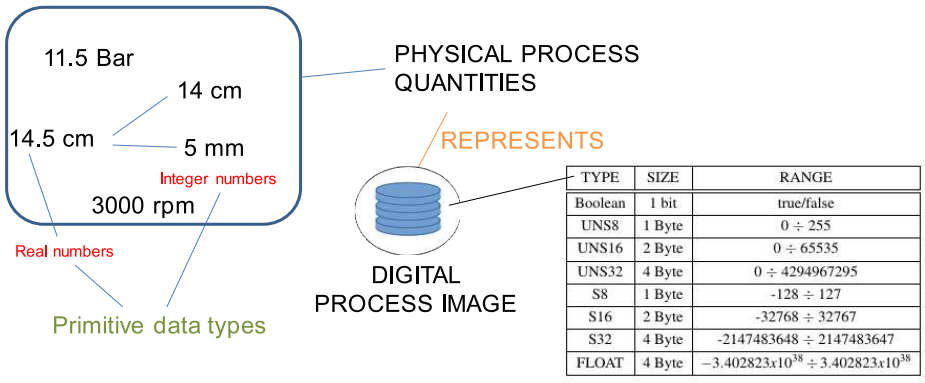


Figure 2.5: Examples of data structure centered systems architectures.

The overall system elements and functioning is modeled in the figure 2.4, in which is represented a generic set of embedded devices which have common firmware libraries and specific implementation for the single device architecture, some of which are shared through classes of them and others like the communication libraries which have to be common to all of the embedded devices. Communication protocol implementation are designed to be supported by all the elaborators units. Using a data structure centered design strategy for the firmware elements grants a data structure which can mask the implementation below provided that the realized behavior is the same. Firmware modules are interfaced though apposite variables that are contained in the communication data structure, and this gives that possibility of having a modular approach to the insertion of features which need the only effort of inserting their variables in the data map. Protocols data maps can have different forms (tables, dictionaries, memory references) that depend mainly on the specification of the protocol, but due to the fact that data exchange is the target, their formalization is intended to aid the insertion of interface variables. The data structure centered design also exploit another important concept which is the base of the entire system architecture. The

digital process image is the shared data between the elements of the systems which contain the representation of the physical quantities that are controlled by the device. The representation of the physical quantities is usually done through data variables which are primitive data types. Primitive data types which are integer numbers, real numbers and logical values are common in all programming environment and are the based of communication protocols data maps.



TYPE	SIZE	RANGE
Boolean	1 bit	true/false
UNS8	1 Byte	$0 \div 255$
UNS16	2 Byte	$0 \div 65535$
UNS32	4 Byte	$0 \div 4294967295$
S8	1 Byte	$-128 \div 127$
S16	2 Byte	$-32768 \div 32767$
S32	4 Byte	$-2147483648 \div 2147483647$
FLOAT	4 Byte	$-3.402823x10^{38} \div 3.402823x10^{38}$

Figure 2.6: Primitive data types and process variables.

So the digital representation of the data is encoded through the definition of these primitive data types which are a common element inside all the elements involved in the framework and this make possible to have a common interface for all of them which is the data structure itself. Fig. 2.7 shows the interactions between data and some of the framework components: The C.I.T application that generate both software and firmware data maps, firmware modules that have data structures which are accessible through communication protocols and S.I.E.G program that generates a complete software program based on these data definitions.

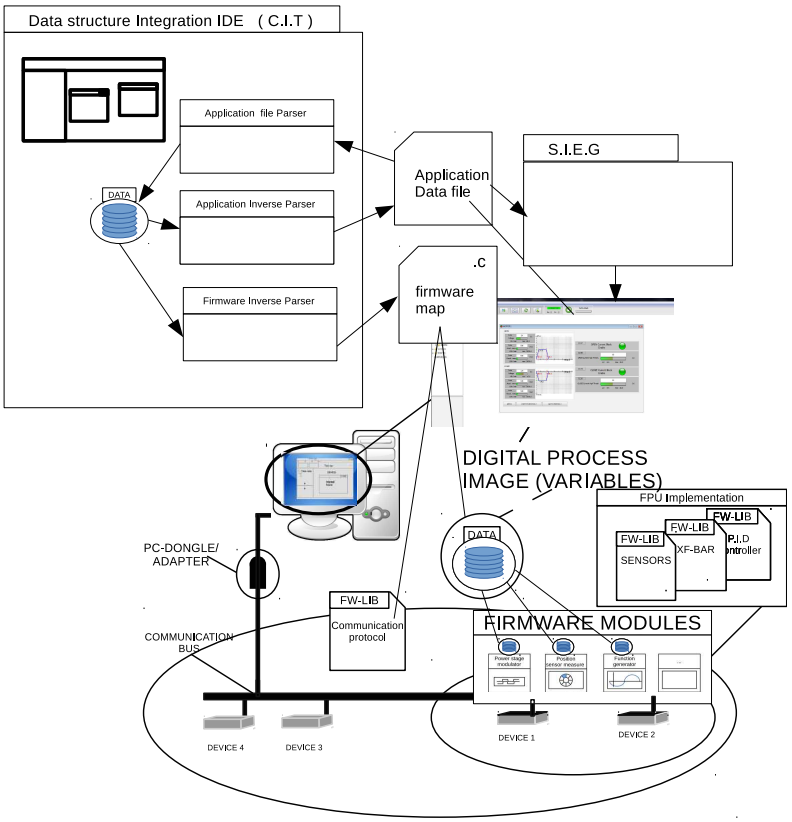


Figure 2.7: D.A.E.D.A.L data centered architecture and data interactions with framework's main elements.

2.2.2 Software

The model in figure 2.4 represents a bus structure that connects all the devices and the software program which is executed on a general purpose PC, that is connected to this bus through a proper adapter or dongle. The PC-Application has to handle the specific adapter and connection ports and then create a digital image of the device features that is independent from the implementation. The device digital image is automatically generated using the Integration Tool data structure which is interpreted by the proper modules that use these informations and the interfacing with the communication libraries. These modules always automatically update the image granting its coherency. Another module is intended to automatically generate the presentation of the image of the device through graphical components which are generated interpreting the same Integration Tool data structure. The details of these processes will be explained in the next chapter. Last but no least there is the integration or configuration tool program which is a graphical editor which is able to compose the data GUI of the software interface program and in which the user can insert, through optimized forms, the data structure of the communication protocols that the device has to support in a very user-friendly and fast manner. Then always through dedicated editor forms it is possible to link the device data to the graphical components. This operation grants that every graphical element linked to a parameter is always updated on every change of it and also every user input is propagated to the parameters and then to all the presentations. Usually the procedure of composing the communication tables, dictionaries or every kind of related structure is done manually or through proper software instruments for the specific protocol. These instruments, which are made especially in the handling of complex protocols, are used to avoid the manual update of the firmware code files that contain the data, but none of them can automatically generate user program views and data handling layer. CanFestival is an example of an open-source project which is a portable implementation of CANOpen executor and supervisor protocol stacks which has a software tool to compose the data dictionary of the target device to avoid the manual compilation of the file. It Is one of the most widespread especially for the fact that is free and open-source, but like the other instruments both professional and not, it does not provide the automatic generation

of a program which is able to interface with the dictionary.

2.2.3 Firmware

The firmware part of the framework is a collection of libraries which have been developed for certain classes of devices. Differently from the software part, rather than the reuse of code, the aim is to have a modular environment. This also increases the reuse of code, but often due to need of specific features the code that can be reused is limited and is better to have a structure which can certainly use what is already done; nevertheless, the important thing is that it should be able to be extended easily to meet device specific needs and to have a significant advantage in the building of optimized modules. In this context there are two different topics to handle: One is the communication layer which should be common to all the devices, at least for the one supporting the same protocols, and the method in which can be achieved the maximum portability of the libraries that are supposed to accomplish this task. The other in the possibility of creating modules which are specific for the device classes that can be easily integrated in firmware programs. While for software the reuse of code is easier and the mechanisms are intrinsic in languages, especially for object oriented programming languages (OOP), in the case of firmware elements the construction of these instruments is not so structured and the building of such structure requires the proper construction strategies that become part of the framework. The development of these methods will be analyzed in details in the chapter inherent to the design of the firmware part of the framework.

2.2.4 Data Coherence: Propagation and Automatic Handling

When dealing with data, coherence is the most important characteristic, without it, data can not be considered reliable, so for systems which have distributed digital images of the same process is essential to grant that this characteristic is always present. This is one of the most complex part of the entire system because it involve both firmware and software elements and their interfaces. Granting data coherence in this scenario make use of two different elements, a known approach which is automatic

device firmware tables and dictionaries generation, and another that is the software design based on the adaptation of known patterns in the development of data dependent modules of the software environment.

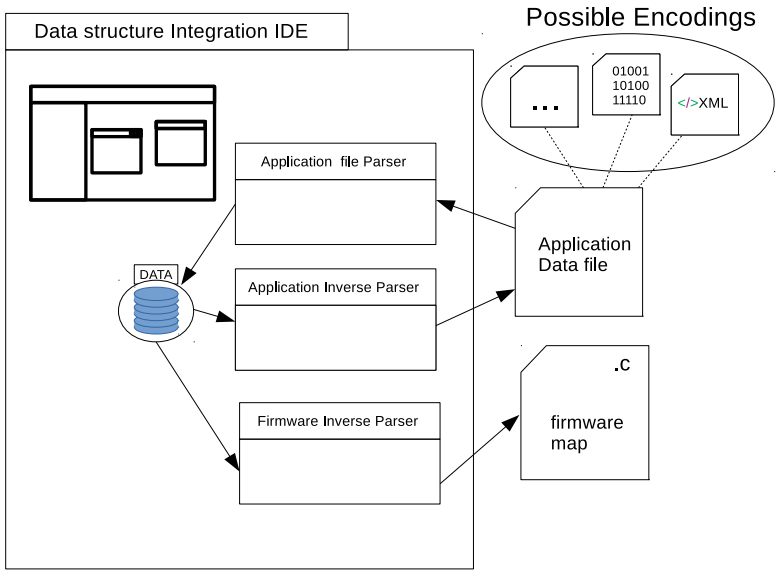


Figure 2.8: C.I.T or Data Integration IDE, application file and firmware source code map generation.

For what concern the first element when the proper data representation can be manipulated through a program editor it is easy to write a module able to compose a text file with the proper keywords to produce a source file with the construction of the relative protocol implementation language containing the data map. Then the file has only to be included in the source projects files of the firmware program and compiled with the others. It is a similar operation to the generation of the software shared data file, but with different encoding rules, as will be shown in the software design chapter this process can be done with the inverse parser pattern which effectively use the same mechanism but with different rules. The generation of the firmware data map

and the software data file, at the same time, through the same instrument, that work on its internal data representation grants that the generated maps are always synchronized, because it is the same data processed with inverse parsers, in which change only the encoding mechanism. Supposing that inverse parsers does not have any malfunctioning, the digital image that they produce is the same. With the centralization of these processes it also possible to control their evolution and trace the versions of these artifacts and easily keep them linked, see fig.2.8.

The architecture of the system adopts methods and patterns which are intended to automatically handle the data consistency. These concepts interest most the communication features of devices. Communication libraries and data handling libraries are designed and constructed with dedicated patterns and strategies built upon some of the already theorized ones.

This architectural design is a response to the need of a common interface between elements which have significant difference in the construction paradigms; software which are OOP products and firmware one which are situated in lower layers of abstraction. This common digital image is also the base to build coherency mechanisms and decouple the logics of the involved element that interact only through data. This strategy does not impose constraints in the software development and effectively decouples the data model from the application components. Some limitations are imposed in the construction of firmware application logic related modules which have to be strongly coupled to an exchange data structure that can be mapped inside protocols exchange mechanisms. This option is not only a constraints and can be used to decouple firmware modules containing application logic from the one which are more related to the handling of the hardware on which these code modules are executed. All these aspects will be further clarified in the design of the framework elements in the next chapters. Fig. 2.9, introduces some of the core elements of the framework that are discussed in details in the next chapters. This figure collects different representations that are encountered in the reading. It is important to keep in mind this figure in the reading of next chapters, in order to understand both the overall concepts and the mechanisms that are behind the composing elements.

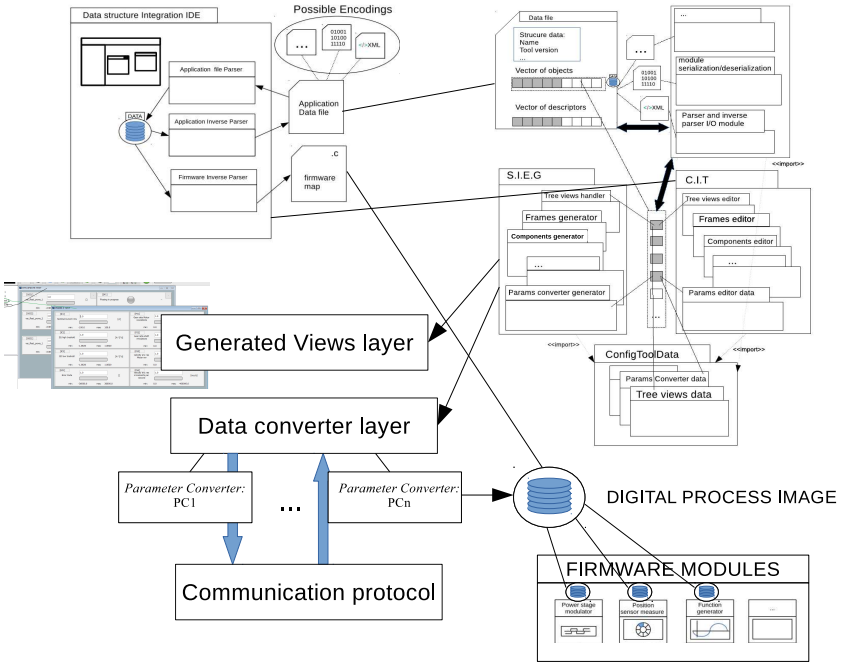


Figure 2.9: D.A.E.D.A.L core elements and patterns: C.I.T and S.I.E.G applications, modular and extensible data structure, data converter pattern, views generation and firmware module design based on data interfaces.

Chapter 3

Software Design for Embedded Industrial Devices

The software design for industrial embedded devices has to accomplish some specific tasks. Devices that are found in industry, especially electric drives, are often electronic boards which communicate with the outside through industrial communication protocols and have a lot of parametrization capabilities. They are also provided with diagnostic features which can give important information about the system status and applications functioning. The software programs that are involved in this field have to support communication protocols to interface with devices and be able to command, monitor and send configuration to them. The features involved are often complex and have to be presented in an easy to use manner. The standardization of protocols and common elements can be a relevant point in order to develop reusable components. This is really important due to the fact that the realization of these kind of programs or only part of it can require years of development.

The general approach in writing a software is to reuse code as much as possible writing libraries, also known as toolkits and frameworks. According to a commonly accepted classification, toolkits are the object-oriented equivalent of subroutine libraries, a framework is a set of cooperating classes that make up a reusable design for a specific class of software. In the writing where these considerations are made and

it is also affirmed that applications(software programs) are hard to design, toolkits, which today are maybe referred more as class libraries, are harder, and frameworks are the hardest to design [49].

This chapter present some of the significant aspects of the development of a framework for the described purpose. For the fact that frameworks need a lot of design efforts, a structured approach has to be embraced, and before entering in the specific writing, a summary on the state of the art of software development is presented. These guidelines should be considered in the realization of every project with a huge potential complexity.

3.1 State of the Art of Software Engineering

Software engineering, is the discipline that concern the many aspects of software development. Definitions from IEEE:

"the systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software" [50]

"The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software" [50]

These are the definitions of ISO standard vocabulary of information technology. Both ISO and IEEE organization agree that the process described in the SWEBOK is the one that should be known and mastered by every software engineer which can be qualified as such. There are fifteen recognized knowledge areas (KAs), which will be briefly described in the following sections, twelve of them, are the ones mentioned in the introduction chapter. They group all the software engineering related disciplines, and the remaining three are the more related with the computer science field of which, accordingly to the book, software engineering is built upon. Computer science deals with the subjects inherent to the development of algorithms, discrete and

binary logic and arithmetic. Every knowledge area covers a group of topics, some of them are considered as specific subjects which can be found as courses of study inside institutions and others do not exist as a discipline themselves. Not all the important and existing computer science disciplines are covered by these KAs, as the case of computer graphics. These areas are mainly focused on giving all the necessary elements involved in software development. The computer graphics example even if not included in the book, need the knowledge of many presented topics only for a first approach, and excluding CAD programs, the presence of computer graphics is more inherent to physical simulators, entertainment, video-games, virtual-reality programs, which development requires a very specific approach, just think about graphics and physics engines. Probably these topics are wide enough to obtain their own "body of knowledge" book. Anyway these topics are the bases also and especially for these kind of programs. Projects that need many of these very specific and vast topics often require the presence of many experts in the concerned fields or developers with highly transversal knowledge and skills. Such situation is hard to have especially in small and emerging companies. This fact, in addition to different schools of thought, has also introduced techniques that are labeled as "Agile" programming, which identify lighter development processes, which is some cases is the only possible way, but in other contexts their excessive use can contribute to the loss of strategical and long-term benefits.

3.1.1 Software Development Knowledge Areas

The following of international guidelines in software development is not an assurance in the realization of a quality product, but at least is a good starting point and using tested procedures helps obtaining certifications. Also software projects can interest multinational domains, so it is important having a common language to organize teams and companies. Is important to distinguish the knowledge areas from the development process itself in which some areas are the core and most critical ones. Design, construction, testing and software configuration managements have a huge impact on the development process because are the more involved in the methods and strategies. Inside these areas there are some of the most expanded and discussed

subjects. One of the most relevant is "design patterns". They appear in the Software Structure and architecture topic, which is the program part that influences every other [49]. As already mentioned these KAs are the more bound to methods and strategies which are specific for instruments and projects and require a significant engineering effort. The foundations of computer science are the subjects which are used for their implementation and the other KAs are intended to support to process. Design strategies and methods can influence the whole activity and the resulting quality of the products, their key role can make the difference between success or failure in reaching project goals. Fig. 3.1 shows a possible exemplification of the roles that KAs cover inside the software engineering process which is a KA itself.

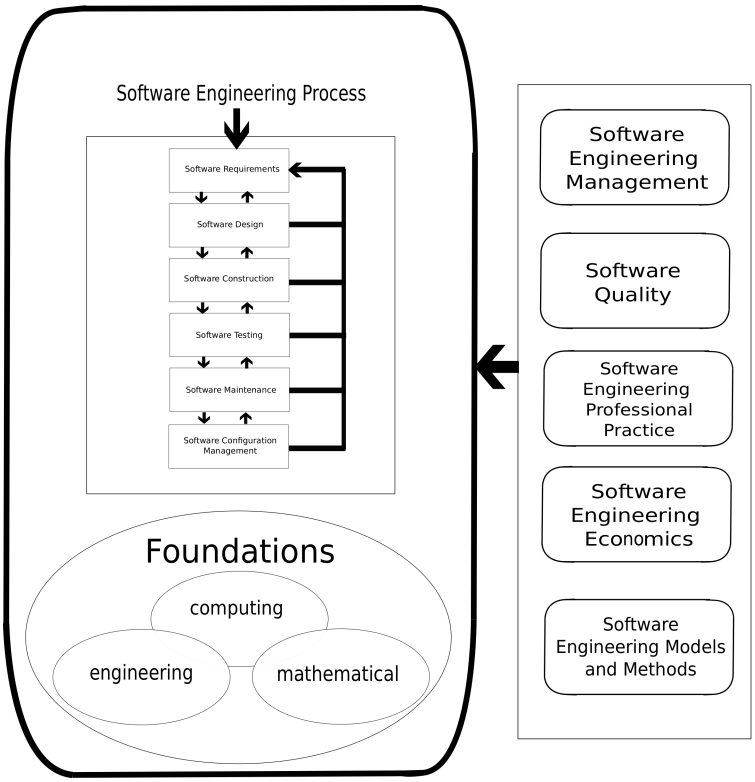


Figure 3.1: Software Engineering Process, Knowledge Areas.

3.1.1.1 Software Requirements

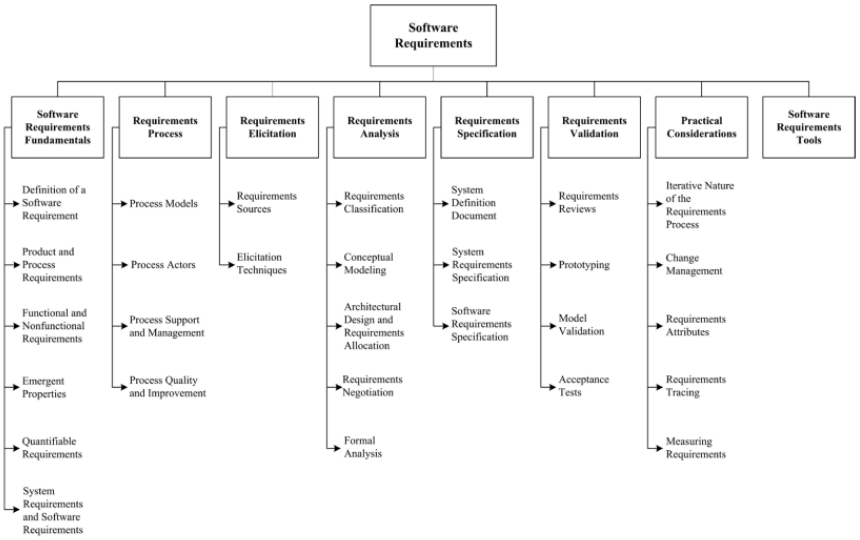


Figure 3.2: Software Requirements topics

Software requirements process is often a difficult task and is always the root of the development process [51]. This stage can interest stakeholders with different extractions, customers, technicians and many others, even end-users, which have different skills and mindsets. In this stage it should be defined what a program has to do, in a way that could be understandable by the different figures involved. It should be also considered how requirements are expected to evolve during the life-cycle of programs. It is also important to trace the meeting of requirements during the design stages and be able to face possible changes. The nature of this process is iterative and keeps following the project until the retirement.

3.1.1.2 Software Design

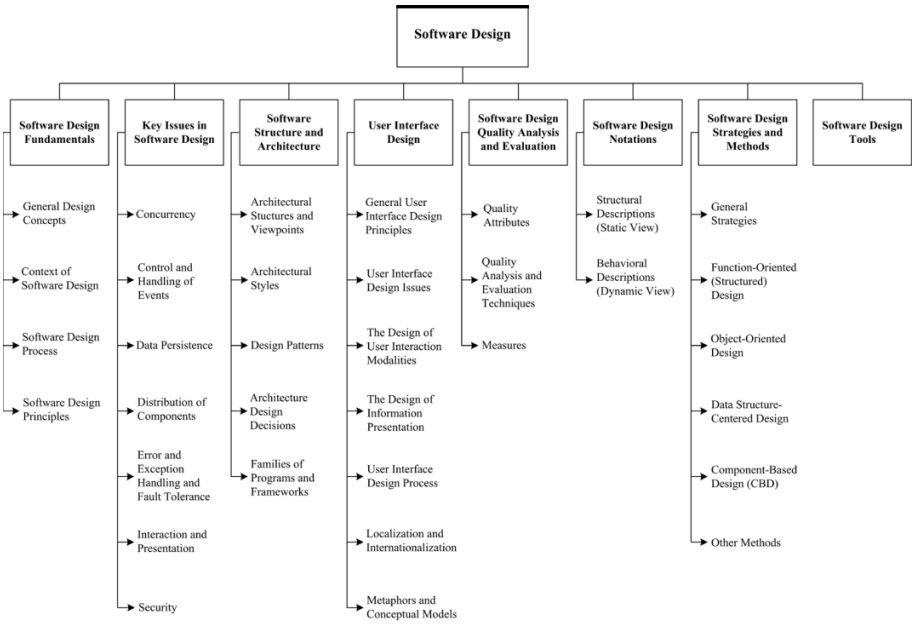


Figure 3.3: Software Design topics and subjects.

The most critical part of a software is its structure, the architecture and general definitions of behaviors, elements, modules and every technical aspect of the project regarding this stage, will define the program from functional to presentation parts. This KA is the core activity of the engineering process.

3.1.1.3 Software Construction

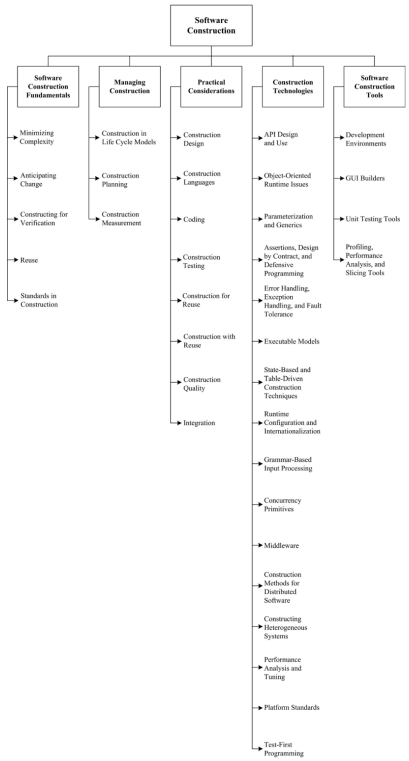


Figure 3.4: Software Construction topics

Software construction consists in the assembling of the program following the design plan. Some topics which are derived from the strategy are so relevant that are considered part of the construction, that is the reuse of code and elements. Due to complexity that a software can reach, in terms of elements, execution flow and size, the construction should not add more of it. Especially in Object-Oriented languages the syntax and instruments aid the reuse of code and the code organization in a more organized and comprehensible structure. Proper construction methods, Component Based Design (CBD) and Object-Oriented Design, see fig. 3.3, can reduce development time through reuse of code.

3.1.1.4 Software Testing

Figure 3.5: Software Testing topics

Testing the software is often a difficult task, and "bugs" are often found by end-users. This situation, can cause disservices and failures. Covering all possibles cases is almost always impossible or extremely time consuming. This operation is also very specific for the kind of services and purposes that a program has. That is the reason behind development of tools that support this stage, which can use different techniques. For example: Model-based, usage-based and fault-based. Strategies that can increase reliability of these products can avoid most of the issues in user experiences.

3.1.1.5 Software Maintenance

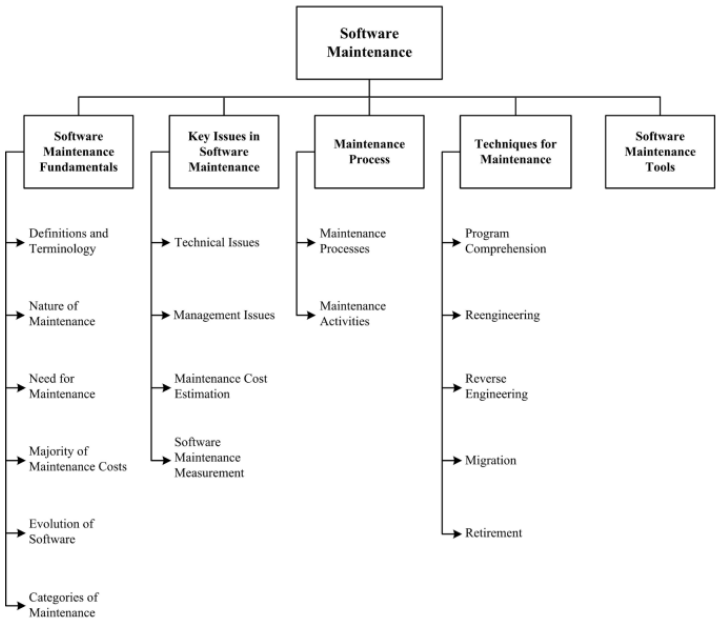


Figure 3.6: Software Maintenance topics

The maintenance of a software does not involve only the error correction and bug-fixes, it is the process that follows the evolution of a software when it is already distributed and used. A program can work perfectly after the testing stage and never shows unwanted behaviors, but could happen that requirements evolve, even due to an evolution of the context of application and then some improvements are to be made. New releases have to be distributed and old program versions have to be retired.

3.1.1.6 Software Configuration Management

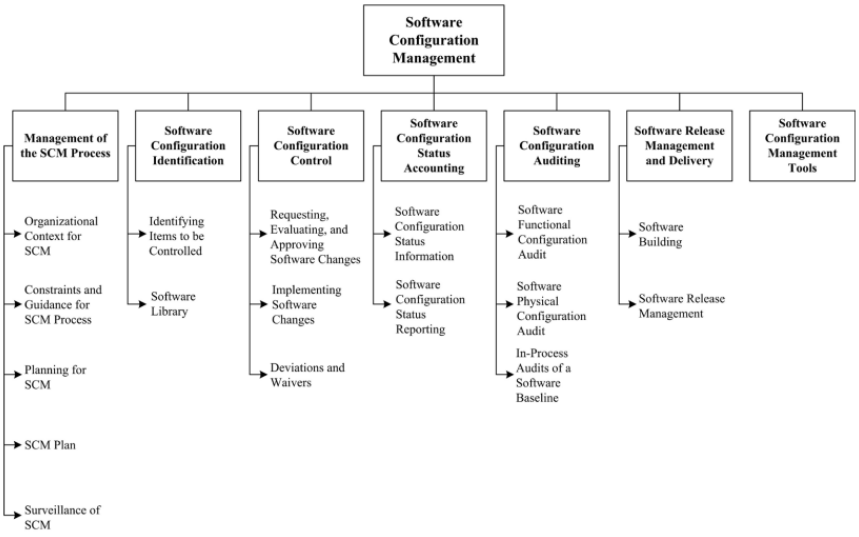


Figure 3.7: Software Configuration Management topics

Software configuration management is maybe the hardest task to face, evolutions can interest both the program itself and also its elements. During the life-cycle of a program a lot of releases can be made which have also different versions of the composing elements. It is important to know on every release, what are the present features and eventually known errors. Configuration can also be made through data and customization of the product. Is important to trace all these configurations and be able to retrieve their characteristics or even the same developers in charge of the project, will not be able to understand what was the old and what is the new.

3.1.1.7 Software Engineering Management

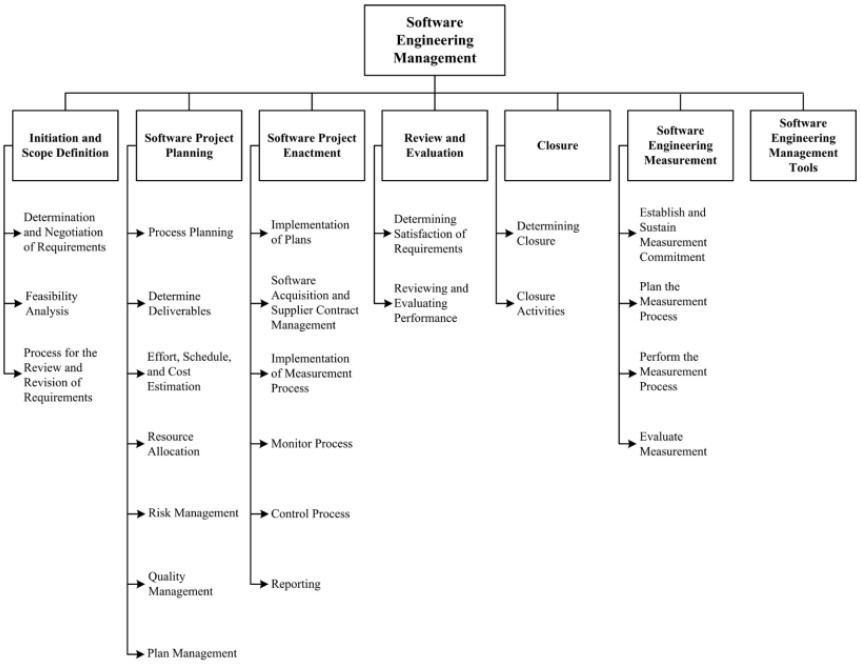


Figure 3.8: Software Engineering Management topics

As every engineering process, there are goals, timings, constraints and resources. The management process should define activities and allocations to proceed with proper and feasible development plans. Every project has its own characteristics that have to be considered and managed by the right professional figures.

3.1.1.8 Software Engineering Process

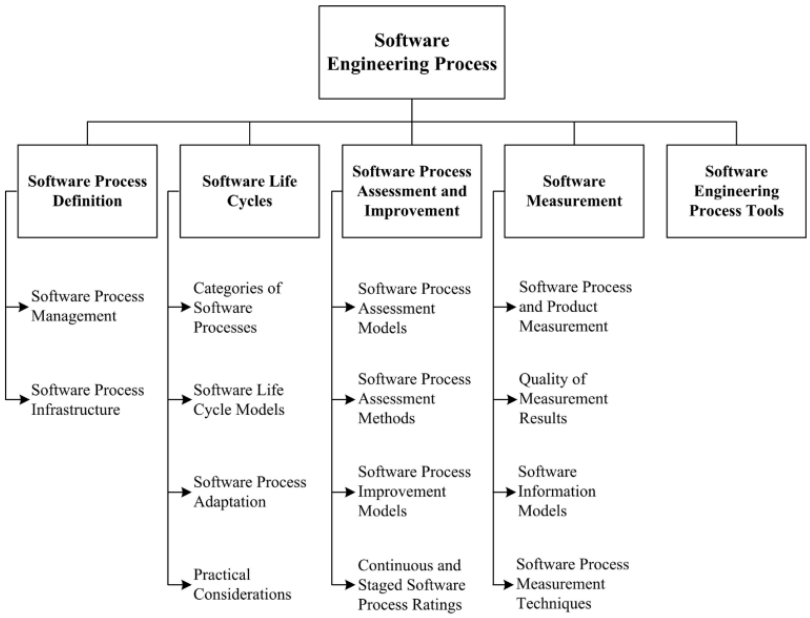


Figure 3.9: Software Engineering Process topics

The topics of this KA are deputed to follow the whole process of software development and life-cycle, evaluating the impact of its evolution and usage. It is like every other engineering process that has to grant the realization of quality products through proper processes and validations.

3.1.1.9 Software Engineering Models and Methods

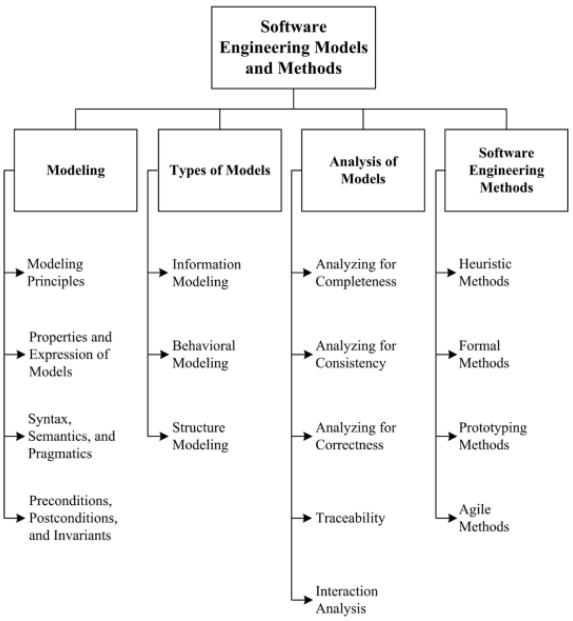


Figure 3.10: Software Engineering Models and Methods topics

Modeling, is one of the most important activities that side the design and construction of a software. As already stated in the introduction, models are the maps of programs characteristics, they are necessary to trace and understands the composing elements and behaviors. Methods are inherent to the practical techniques and strategies applied to the instruments used for they construction.

3.1.1.10 Software Quality

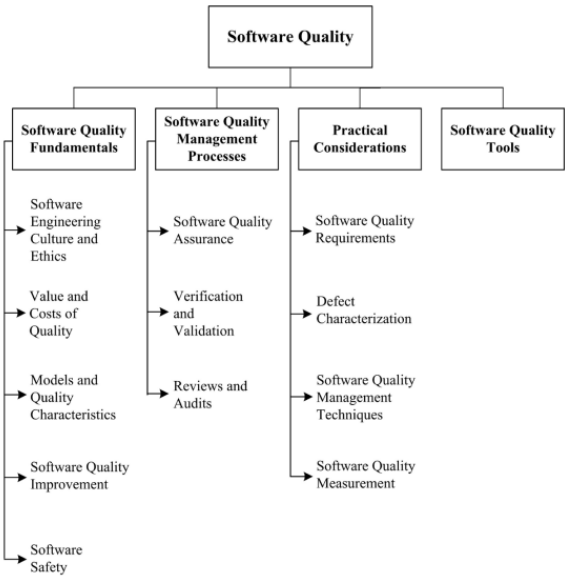


Figure 3.11: Software Quality topics

In defining the quality of a program different factors are involved, from safety to functional aspects and also cultural. Inside software development there are extremely technical tasks, but also some humanistic traits in writing program code. The understandability of source code, which is always an important trait, is a characteristic with a highly human-subjective component. In writing understandable code there are of course some rules, but also a lot of conventions and possible interpretations of guidelines.

3.1.1.11 Software Engineering Professional Practice

Figure 3.12: Software Engineering Professional Practice topics

Is important to remember that engineering is a professional practice in which there are a lot of implications, ethical and professional conduct are part of the work. Software engineering involve a lot of team-work, even when there is only one developer, because even in that case, the project will quite for sure use code and libraries written by other programmers. A sentence that is well know by the most of programmers is: "write the program code as someone else should be able to understand it". That is because probably someone will have to.

3.1.1.12 Software Engineering Economics

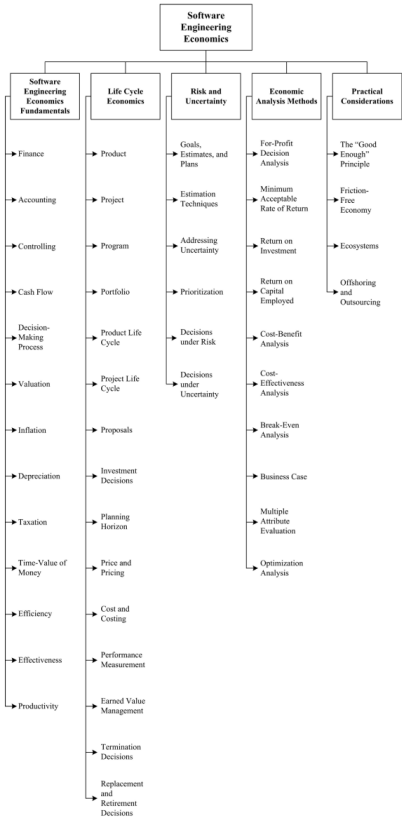


Figure 3.13: Software Engineering Economics topics

In a professional activity, the economic factor, luckily or not, is part of the process. Engineering economics is deputed to the rightful management of the financial part of the project.

3.1.1.13 Computing Foundations

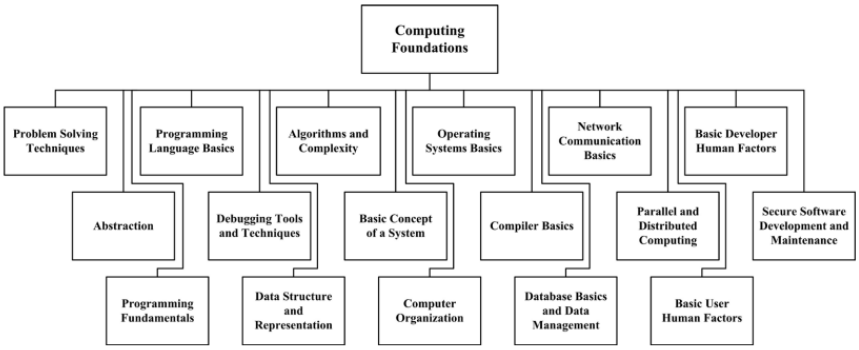


Figure 3.14: Computing Foundations topics

Foundations are the base disciplines of software engineering, the computing ones, include all the necessary skills in using computer programming to solve problems. Disciplines are inherent to the different information technologies that are used in the software environment. This is a vast KA, that ranges from writing a single line of code to the world wide web.

3.1.1.14 Mathematical Foundations

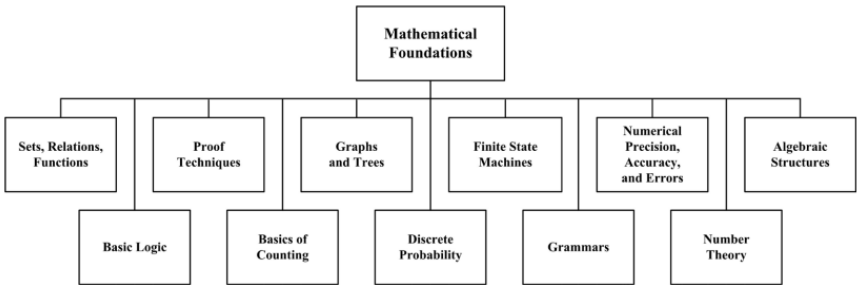


Figure 3.15: Mathematical Foundations topics

The mathematic used in computer science has proper formalizations to handle the mechanisms that are intrinsic of computer processing, derived from the digitalization of quantities and the fact that elaborations are done through execution flows, which have state based logics. The logics involved in computer programming are different from pure mathematical forms, because software logics can evolve inside the software itself, such as some disciplines inherent to artificial intelligence, like knowledge based domains and machine learning.

3.1.1.15 Engineering Foundations

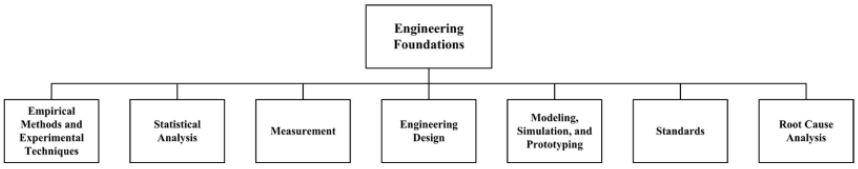


Figure 3.16: Engineering Foundations topics

Engineering foundations are the concepts which are present in every engineering process. Design activities, quality measurements and products handling.

3.1.2 Agile Programming

There are other strategies which do not follow a structured development like the one proposed above, these techniques are referred to as "Agile". They are often termed also "anti-pattern" which identify a category of guidelines less structured with the purpose of developing program elements to meet requirements in the shortest time possible. There is also a manifest of the agile programming [52], which at the moment sums up this procedure in twelve principles:

1. *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*
2. *Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.*
3. *Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.*
4. *Business people and developers must work together daily throughout the project.*
5. *Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.*
6. *The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.*
7. *Working software is the primary measure of progress.*
8. *Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.*
9. *Continuous attention to technical excellence and good design enhances agility.*
10. *Simplicity – the art of maximizing the amount of work not done – is essential.*

11. *The best architectures, requirements, and designs emerge from self-organizing teams.*

12. *At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.*

Agile programming has led to the diffusion of different methods like extreme programming (XP) [53], scrum, feature driven development (FDD) [54] and rapid application development (RAD)[55],[16]. Such strategies are always based on continuous iterations with stakeholders to analyze the advancement in the requirements achievements. They can be used in some context especially for projects with limited complexity.

3.1.3 Design Patterns

Patterns, not only the design ones, are the base and one of the most explored topic of the engineering processes [56]. Their diffusion has interested in particular the object-oriented world and even if it was a concept that was present since the beginning, one of the most significant contributions to this subject is the book "Design Pattern" [49], written in 1996 by the "Gang of Four" (Go4) which are very notorious in the history of software engineering. Patterns are the response to many problems that can be encountered in the development of applications and cover many aspects:

- Creational.
- Structural.
- Behavioral.
- Concurrency.

The actually known and diffused patterns are mainly based for object oriented paradigm due to the use of the instruments provided by these languages which are inheritance, composition, encapsulation, delegation, interfaces, polymorphism, reflection and open recursion. They are applied in many software environments and even there

are schools of thought that are totally against the "pattern philosophy", these guidelines have been applied in several projects and have led to significant benefits. Of course their excessive and erroneous usage can be inappropriate and can justify the criticism on the subject, but as will be shown also in this work their correct usage can bring to general solutions that solve specific problems with a lot of benefits.

3.2 Software Design for the Electric Drives Framework

Why the need of a software framework for Electric Drives? Working on different projects in last years both as student and sometime as consultant, it happened to face the development of programs to interface with embedded devices, sometimes with very different application, see appendix B. What can be seen is that the development of these programs has many elements that are repeated even if the devices are different. Data handling and communication tasks are very similar, but not identical, so a good strategy to increase the reuse of elements and reduce the development time is to create a flexible structure based on this common ground. The developed framework, has been introduced to the category of electric drives as a case study, but the methods can be used for every embedded device which is able to communicate through at least a communication protocol. A significant role is covered by software programs with GUI, the commissioning ones, which are able to interface with devices through communication buses. The development of such kind of application requires many of the areas of the engineering process, GUI construction is a laborious tasks that requires the writing of a lot of lines of code, image processing, geometrical compositions and user input handling. Building a graphical user interface, protocols stacks, applications data structures, application logics are activities which require the writing of thousands lines of code. Libraries can increment the reuse of code but even the change of a parameter in the device logic would imply the modification of the program code from communication routines to GUI elements. S.I.E.G.Fr.I.E.D has the specific purpose of creating such kind of application and actually is able to create a complete software with graphical user interface and communication features through standard protocols, such as the Modbus that will be used as reference for

the presented examples, without writing a single line of code. Using the C.I.T editor which is composed of forms and "drag and drop" graphical editors, it is possible to define data maps, compose rich GUI, not only without writing code, but also without having knowledge of the complete protocol mechanisms (it is only necessary to know the indexes where variables are mapped and the corresponding type and collocation), and also without knowledge in softwares and GUIs implementations. Of course as every instrument it requires a minimum knowledge on how to use it. The most interesting result is that a person without software engineering skills can compose a functioning programs and is also able to do that with a drastic reduction of development time even for the case of programs written by expert engineers. This permits to focus development efforts and design activities in the improvement and addition of new features minimizing the most repetitive tasks which do not require particular design and innovative construction solutions. This is the purpose of the framework, less laborious tasks and more time for conceptual work, not the substitution of the expertise of people which is valuable and should be used properly. These kind of solutions have always an ethical and professional impact (that is recognized as a knowledge area inside the SWEBOK), and in my opinion it is important to remind this aspect that should be always considered to add values in our lives. Focusing again on the aspect inherent to the design of these instruments, after the little digression about ethics facets, the result claimed above will be illustrated through usage examples and also will be clearer how these instruments work and how they are implemented in the reading of next chapters that describe its internal composition. The programs and libraries that compose S.I.E.G.Fr.I.E.D are written in Java language and make use of objected-oriented paradigms and methods. The Unified Modeling Language (UML) which is the most diffused and specific modeling language for these kind of languages will be used in some descriptions of these mechanisms. UML diagrams will be used to aid the understanding of the most relevant concepts behind the realization of this framework. UML can be used for the modeling of elements and behaviors which can be applied to any object-oriented language, which makes design strategies extremely relevant, because they can be implemented with many different instruments and languages that are based on the OOP paradigms, and not only the Java ones which are

used in the actual implementation. Through this kind of applications it is possible to:

- Configure the device functionalities and parameters.
- Monitor the quantities that are elaborated.
- Create and store parametrization files to assemble specific application configurations.
- Emulate and test the behavior of executor units (slaves) and also communication protocol.
- Emulate, create and test supervisor (master) applications.

The design of these programs which are needed to accomplish the above tasks requires significant efforts and cost in the development. The usual way in which these kinds of application are realized is:

1. Develop or buy the protocol stacks for master and executor.
2. Stacks are usually provided with tools that aid the mapping of parameters and variables for the executor program.
3. Develop the GUI components and organization.
4. Through the communication stack compose the communication tasks and routine to exchange data.
5. Link data to the application architecture and GUI using the strategy which suits more.

Differently from the traditional approach, with this framework the reuse of code is maximized in a way in which a software application can be provided, at least, with basic functionalities to read and write data, without writing any line of code. The extension of the framework features requires of course the writing of new lines of code, but as will be shown, with the S.I.E.G.Fr.I.E.D approach the introduction of new elements is aided by the infrastructure itself and once it has been developed,

it will be possible to reuse it in the same way as the previously developed ones. It is easy to understand that the possibility of having a complete and functioning application of that kind, only inserting data using some guided forms without the need of knowing the protocols mechanisms and software engineering subjects, is an innovative way to reuse code that gives enormous advantages, and how this is possible will be explained in this chapter. The peculiarity of the present embedded world is the strict cooperation between software and firmware elements, that are involved in the realization of a service or functionality, which can be called a module. This has been already explained in chapter 2, and due the higher abstraction layers that belong to the software part, the approach can be more generic, while going down on these levels will be shown how these techniques became more specific. For what concerns the software parts involved, the purposes of S.I.E.G.Fr.I.E.D are:

- Automatic generation of a Software with a Graphical User Interface (GUI) through the interpretation of a data structure realized through specifically designed editor.
- Automatic handling of communication protocols and device parameters.
- Provide instruments to monitor and perform diagnostics where they are present.
- Possible customization of the software interface program without the need to modify the source code, but only data structure.
- Trace versions and customization automatically though the data structure properties and information.
- Create an environment in which the enrichment of feature and GUI elements is made through a modular guided process that minimizes the introduction of errors in the development.
- Generation of both the code part related to the firmware data maps and the application communication through a centralized entity that keeps them synchronized.

The following section presents the details of this innovative approach and the developed programs that implement it. The writing will not cover all the software engineering process which is not the aim of the thesis but some points will be shown in order to understand what claimed above, with particular interest for the design and construction aspects which are the core activities of the development.

3.2.1 Use Case Analysis

In order to understand the effective functioning of the system, a use case scenario, which can be considered a sort of requirement elicitation of what these programs have to do, will be shown their usage through use cases diagram and programs screenshoots. The use case diagrams enclose all the editor activities that a developer has to follow to generate the complete application. Starting from the data insertion and then the graphical user interface composition and navigation structure creation, every activity is guided through an optimized graphical editor. Then the operation of linking variables with GUI components is again very easy, just a matter of opening a window and selecting the variable through from a list. The process ends with the production of the file which is passed through the S.I.E.G that takes this input and produces the complete software as output.

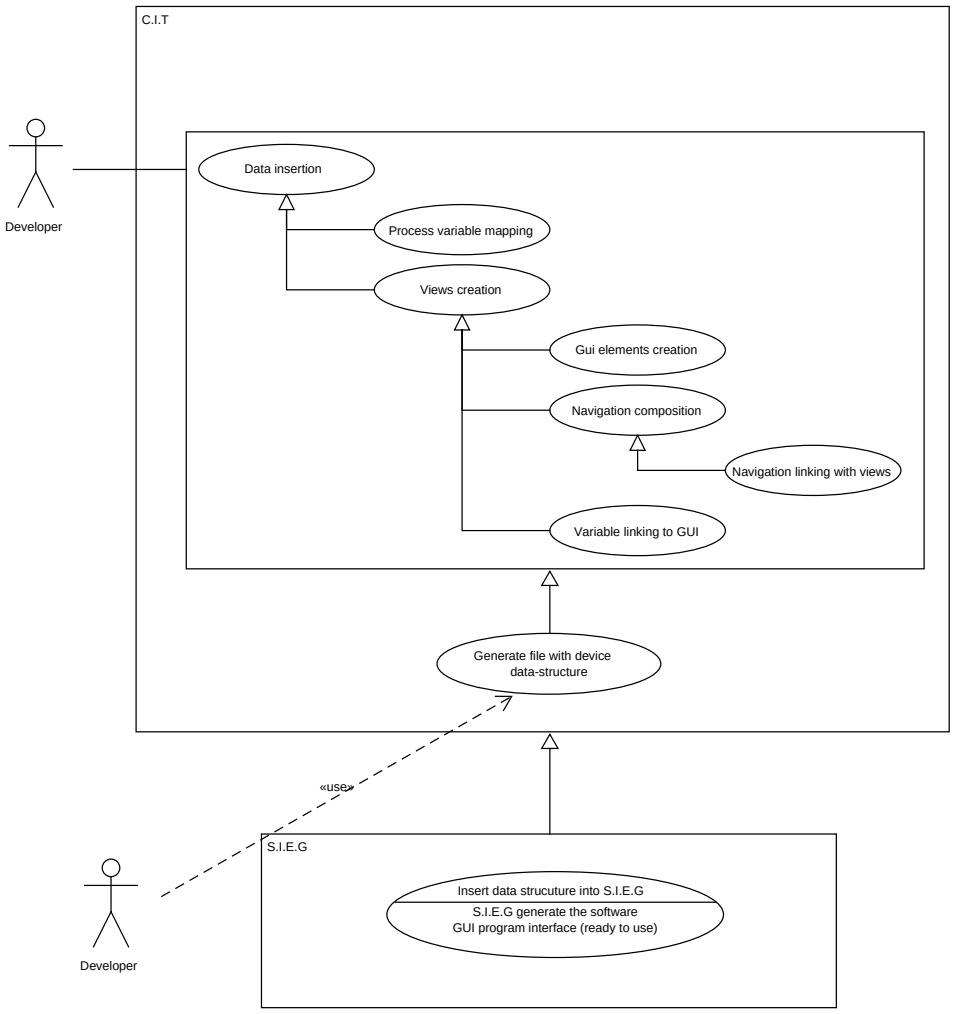


Figure 3.17: S.I.E.G.Fr.I.E.D programs use case diagram analysis

3.2.1.1 Devices Data Mapping

Devices share data through a communication bus between them and supervisor units usually using primitive data-types due to the fact that embedded devices can have very limited resources in terms of memory and this permits to achieve the maximum compatibility and implement it even in the simpler ones. Industrial communication protocols use different physical media and paradigms which depend also on the hardware channel capabilities, but regardless of this, data that are exchanged through the flow of bits are based on the same standards. Primitive data-types is the name used to identify the data strictly related to the binary logic of the elaborators that are composed of few words (8 bits grouping). Some protocols send raw words data and other support more types.

TYPE	SIZE	RANGE
Boolean	1 bit	true/false
UNS8	1 Byte	$0 \div 255$
UNS16	2 Byte	$0 \div 65535$
UNS32	4 Byte	$0 \div 4294967295$
S8	1 Byte	-128 $\div$ 127
S16	2 Byte	-32768 $\div$ 32767
S32	4 Byte	-2147483648 $\div$ 2147483647
FLOAT	4 Byte	$-3.402823x10^{38} \div 3.402823x10^{38}$

Table 3.1: Primitive data types encodings (excluding formats longer than four bytes)

There are other primitive data-types that are constituted of more bytes (double precision numbers, long integer etc...) but their usage is not very popular at the moment, also float representation is often rare to encounter except for devices with the FPU units which use is rapidly increasing, as the arm processor based solutions with the FPU that is being adopted by many micro-controller manufacturers. The reason of the lack of diffusion of these numerical formats till now, as already discussed in the introduction, is the heavy computational cost that has the emulation of numerical

formats that are not supported by the hardware elaboration. Despite that, protocol data exchange interest mostly primitive data-types and the support of numeric formats is not always considered because they can be obtained through composition of other primitive types or simply though bytes concatenations. Protocol mechanisms transfer only flow of bytes which have to be interpreted accordingly to protocol specifications. For example the Modbus protocol does not provide specification for the formats in which bytes should be interpreted, it only transfers 16 bit registers which are used to store data and then it is the application logic that hase to encode them into variables. There are protocols which are more complex like CANOpen which also include numerical formats definitions and their interpretations, but they always use primitive data-types collected in bytes structures which are always exchanged through packets of bytes. The relevant fact which is useful to the design strategy that automatically handles these bytes is that process quantities are always represented through these primitive data-types or some encoding based on them. For example, the string representation through ASCII characters is based on the association of an integer number to a graphical character. This kind of encoding is useful only for the presentation layer because devices processors work with numbers, not letters. Characters are only a graphical way to represent data in a more comprehensible way for the human user.

The traditional procedure to establish a communication between a device and a supervisor unit is to compile the protocol stacks source files which contain the data structure of the digital image of the quantities that are shared, see fig. 3.18. Some companies that write protocols stacks have tools that generate automatically these source files of the device implementation through the insertion of the indexes of the corresponding variables. Usually supervisor units compose the needed functions to access the interested data through the APIs of the IDEs with which they are programmed. These operations always relay on the protocol functions code implementation of the mechanisms which have to be known by developers in order to grant the correct access. The S.I.E.G.Fr.I.E.D approach builds another layer that resides between the software protocols implementations and the application. The software tool takes as input indexes, variables and some other data which allow to choose the

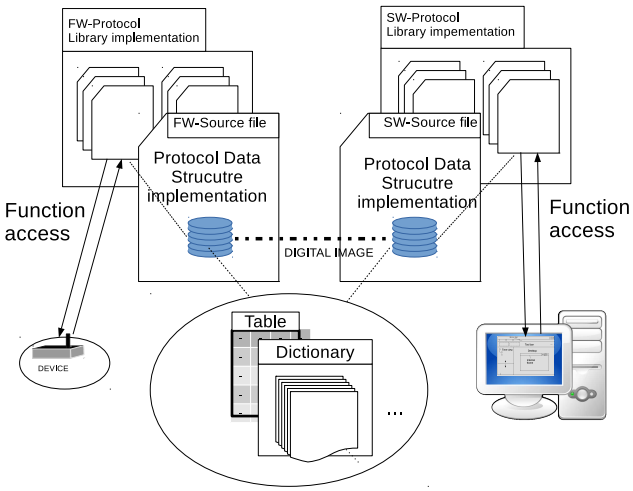


Figure 3.18: Software and firmware example structure of generic embedded devices with communication capabilities.

presentation of the GUI and variables, so in addition to the handling of the firmware data-map it composes a software data-map which is not a static code file but a dynamic data-file which generates the corresponding digital image and is also used to handle directly process variables objects rather than using protocol functions and raw data.

3.2.1.2 Device Map Insertion

The C.I.T GUI is designed to insert and associate variables into protocols data structure. Panels, elements and presentation are optimized for the protocol mechanisms and data specifications, as shown in the picture below where is the editor for the Modbus protocol. At the moment, supported primitive data-types are the ones listed in table 3.1, but the modular structure of the editor easily permits the addition of other primitive data-types. Longer data has not been yet contemplated due to the fact that memory is still a limited resource for embedded devices and also the available ranges are enough for all the encountered applications. The possibility of adding new

data-types has been considered in the design to meet eventual future improvements. The details about this aspect will be shown in the construction aspects of the C.I.T. The C.I.T is subdivided in two parts which are presented in two tab views of the GUI. The first one is the parameter view which is used to insert tables, dictionaries or data-structures information of the protocol or protocols used, establishing where the process variables are mapped. These variables are associated to the digital quantities that are controlled by the device. For example creating a mapping for the integer data of 32 bit that has been named speed, representing the speed of a motor in the indicated measure units, immediately defines that process variable. The second one is the window editor view, which is used to compose the GUI that will be generated through the S.I.E.G software. In this stage is required only a basic knowledge about protocols data-structures and the insertion procedure is totally guided and aided by the program and can be done even by inexpert users after a brief explanation of the task.

There are only two relevant rules in this phase:

- Every process variable has to be associated to an unique identifier string code and the editor shall check for already assigned identifiers (removed variables do not remove identifiers in order to maintain the compatibility with previous configurations).

- Every inserted variable shall be mapped at least in one protocol, that means that a process variable "has value" only when it is accessible by a communication channel (otherwise there is no need to add an unreachable variable in a communication mechanisms).

The C.I.T informs the user if a violation of these rules is made and integrate all the necessary controls to grant a correct data insertion. When a variable is created through the assignment of a name, identifier, mapping and additional format informations, the data-structure is updated with the new element.

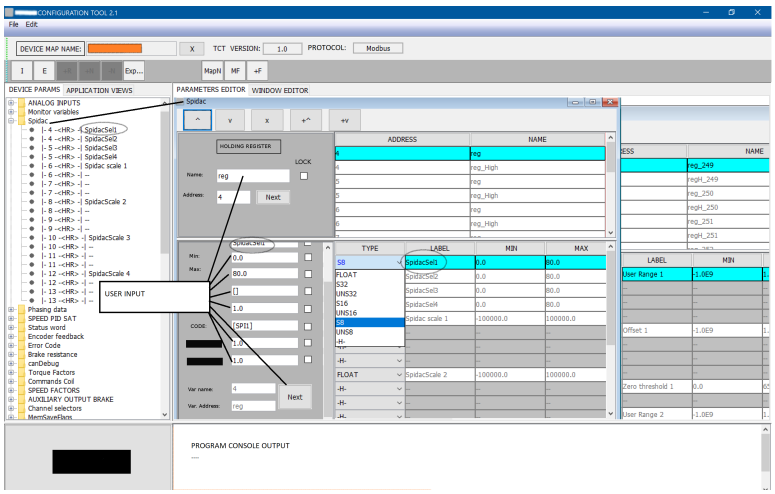


Figure 3.19: Application view of parameters mapping procedure for Modbus protocol through the C.I.T editor.

3.2.1.3 GUI Configuration

Through the window editor it is possible to create the device views of the software interface that will be produced. The views have a common layout, see figure 3.20, which elements are related to the features that every device has, like functions to read all variables, write configuration on device, load configuration from file, save configuration on file, connect to the device, etc... The configurable GUI elements are frames, navigation structures and renderings. These are all the necessary elements to compose a structured GUI based program, but in case of need it is possible to add editor elements to the framework extending its structure without losing the compatibility with previous versions of the software. This is possible thanks to the modular design of the architecture of the framework.

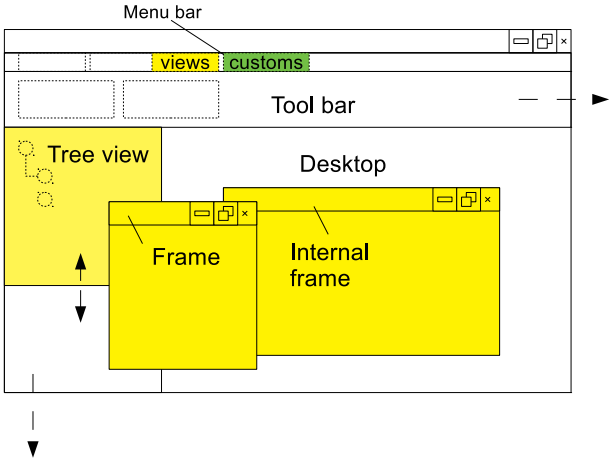


Figure 3.20: Layout composition of the generated softwares, in yellow the elements that can be composed through the C.I.T editor environment, in green the menu item containing features developed using framework mechanisms and APIs, without using editors.

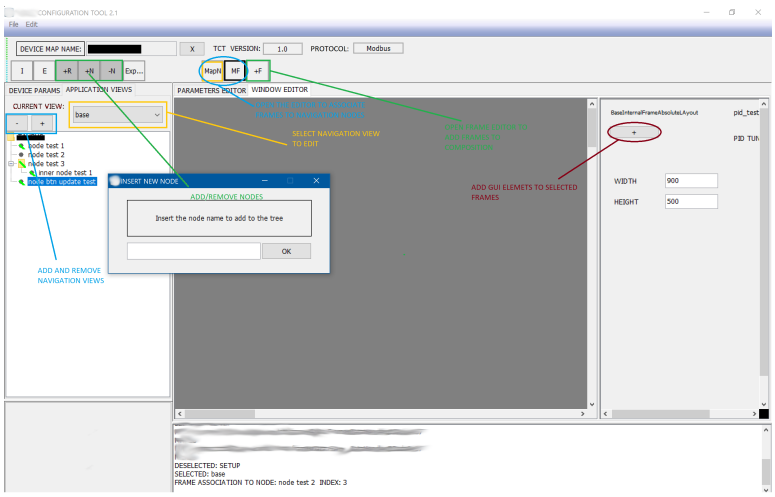


Figure 3.21: Application view of the C.I.T program editor for graphical components.

In the above picture fig.3.21, there is the main frame of the C.I.T with some indications about the interactive components that compose the "window view". Through these tabs it is possible to organize and add navigation trees and access to the various editors for the graphical options.

3.2.1.4 Frame Creation

A dedicated editor for frame creation lets the user choose different windows based on the standard GUIs layouts that are implemented in most of the graphical libraries of the languages which provide support for graphical software interfaces.

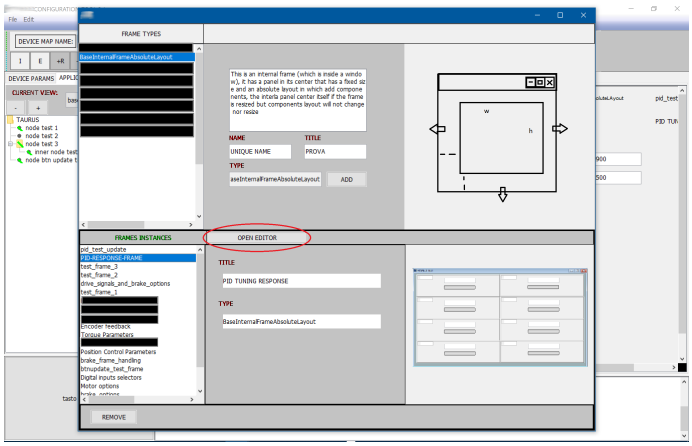


Figure 3.22: C.I.T application, frames editor view.

The editor has a list with the implemented options and offers a preview of the selected layout. In the lower part there is the list of the already generated data elements representing frames. Selecting them it is possible to open the corresponding editor to modify the internal composition of the elements that are used to present and access process variables. Every frame instance must have a unique name which is displayed in the instances list. Selecting an existing instance the editor shows some information about the associated data and its actual composition.

3.2.1.5 Presentation Components Creation

When a new frame layout is generated it is possible to compose the components view that will be presented. Every layout has a dedicated editor with dedicated components that match with the behavior and constraints of the layout adopted. Every layout has

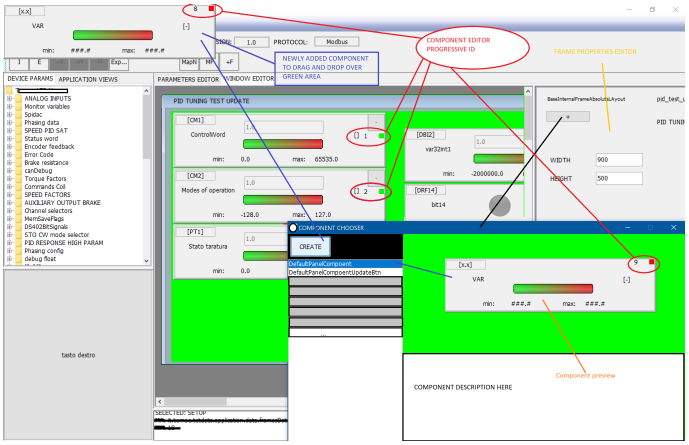


Figure 3.23: C.I.T application, component editor view and frame composition through component "drag and drop".

some properties that can be modified to obtain different flavors and a set of components that can be chosen. Every component type has its own graphical presentation, some provide user input capabilities for read and write variables, and others are just indicators for read only variables. Every component has its own editor, accessible through a right click on it, and a set of properties to customize the aspect and behavior. The composition process is very simple thanks to the "drag and drop" editor implementation, once a desired component is created, it is placed on the screen and it is sufficient to drag it in the desired position inside the layout area.

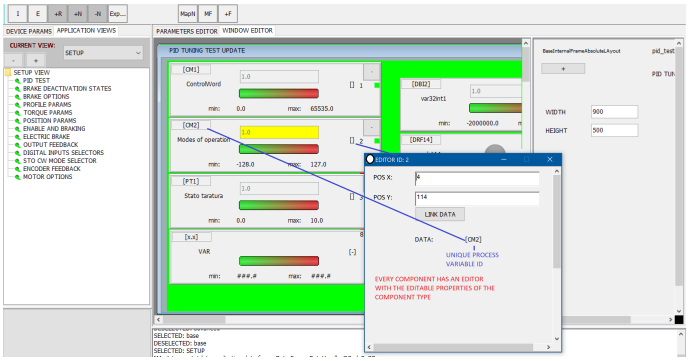


Figure 3.24: C.I.T view, component properties editor and mapped variables linking.

3.2.1.6 Process Variable Linking

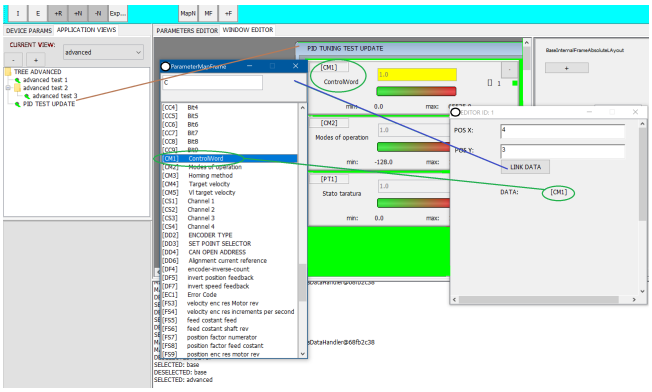


Figure 3.25: C.I.T view, variables linking with graphical components.

Every component editor has the possibility to link one or more process variables, generated in the device data mapping stage, to link their values to a proper user input relationship, behavior and presentation. Components can have the different forms and actuate many kind of elaborations on the user input. The association of a process variable to a component requires only a click on the link button which makes pop out a list of the previously generated variables with a form to select the desired ones and then, link them to the component. There is no limit on the views that can be generated

and the components that can be linked to the same variable. The framework mechanisms grant that all the views are synced with the corresponding variables granting coherency through the whole system. Granting data coherence and consistency is one the most critical tasks in the entire information technology field and especially in the design of quality and reliable software. In the sections related to the design and construction of this framework it will be shown how its internal mechanisms are able to grant this condition.

3.2.1.7 Views Linking with Navigation Structure

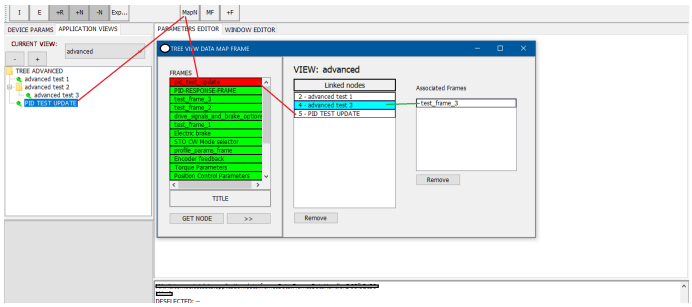


Figure 3.26: C.I.T view, frames linking with tree structures.

When views and maps are added to the data-structure is possible to associate the node of the navigation structure to the desired views. This information will be used by the S.I.E.G program to generate the exploration tree of the application views through which will be shown the selected views inside the application program.

3.2.1.8 View Example

Though the C.I.T it is possible to edit the data-structure at any time and any modification of it will generate the corresponding modification in the generated software interface.

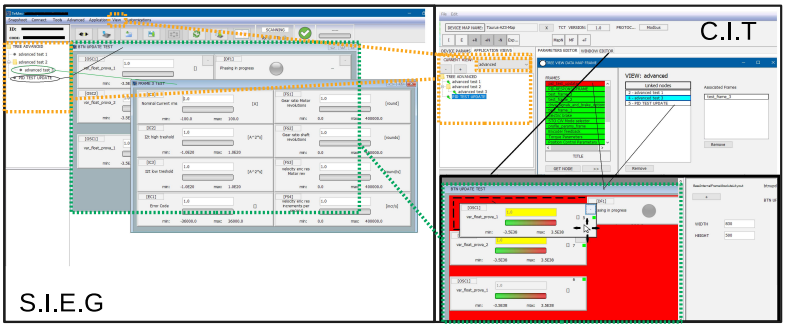


Figure 3.27: C.I.T view, frame composition and generation example.

3.2.1.9 One Program More Devices

The user can generate a file data-structure for every new device that a user want to control, then the file can be added in the proper directory of the S.I.E.G program and used to generate software interfaces which are able to connect to the corresponding devices. Figure 3.2.1.9 shows an example of a software application realized through the use of S.I.E.G which is the software interface for a line of products made by TeMec drive. This approach gives several advantages because, especially in the handling of some products variants and customization, can grant an intensive reuse of code and also can reuse previously generated data structures drastically minimizing development efforts.

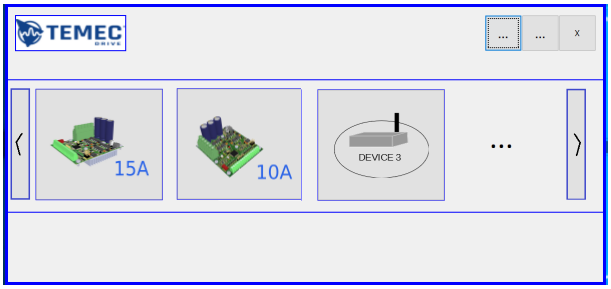


Figure 3.28: S.I.E.G generated view; example of a prototype of a new generated TeMec interface for the AZ2 drive family and any other device that can communicate with industrial protocols.

Fig. 3.2.1.9, shows a prototype of a "device chooser" of the generated software application. Through identification commands provided by protocols specifications, it is possible to handle the selection automatically, but this example helps to understand that also old and new devices can be added, thanks to the protocols standardizations (especially when device source code is available and also the source code of the data map can be used). Without exploiting the code generation feature is possible anyway to generate a complete application through the insertion of devices data maps, that can be found inside devices manuals. Adding new devices or extending the features of an already developed one can be done entirely inside the C.I.T development, which

can easily modify existing data-structures that represent the digital image of the exiting devices. This is also extremely useful during the development of new products because in addition to the monitoring and control of the desired variables it is very easy to add others for debugging and create configurations for that purpose and then they can be refined for the release. Data configurations can be easily stored and versioned to keep trace of the development process. This approach can produce an entire software application with a GUI rich in features in few hours of work. Realizing the same software even using libraries and maximizing reuse of code can take months. With the increase of C.I.T features the development time continues to drop down. At the moment there are many other useful features which are in part or completely not yet implemented, but the first results in real applications are very interesting. How this result has been achieved will become clear in the analysis of the design of the framework that will be presented in the next sections of this chapter. The proposed use cases show a solution that is able to reduce development time and maximize the reuse of code through a paradigm that can be called Construction by Configuration (C-b-C) [57]. The approach has not only these purposes, but there are many other issues that have to be handled in the life-cycle of software products like configurations and versions management and quality related ones. In the next sections will be also given a view of the mechanisms which also improve the robustness of the software applications, generated by the framework, towards these issues.

3.2.2 Software Architecture

As introduced in the previous chapter, there are two programs which are the core of the framework: the Software Interface Environment Generator (S.I.E.G) and the Configuration Integration Tool (C.I.T). This subdivision offers some relevant advantages: The software that generates the interface is the only one that is distributed and used to connect to devices by end users, it has only to interpret a data structure and compose its presentation. The building of this structure is totally left to the C.I.T which is not distributed, so quite all the code related to the generation of the model and inherent to the software engineering process, is protected against de-compilation attempt to copy or manipulate the program, the violation of the S.I.E.G will not achieve all the functionalities needed to have the features listed above. This advantage should be the less important because intellectual property is protected by international laws, but industrial espionage is a real problem, so especially for companies that invest on these instruments having some additional security can be an added value. The most relevant advantage is that this interaction between the two programs has to follow a precise development process both for using the already present features and also for extending them.

There are only few modules which are shared between the two applications, which are inherent to the data structure that is generated by the C.I.T and interpreted by the S.I.E.G application to generate the GUI elements: The InterfaceGUIGenerator and the ConfigToolData are packages which are used to build the corresponding libraries and be used by the applications. The first collects all the classes that generate objects, such as GUI elements and data converters, the second one contains their data model. In the above figure 3.29, there is another important element which is highlighted in green, that is the module which contains the editor components.

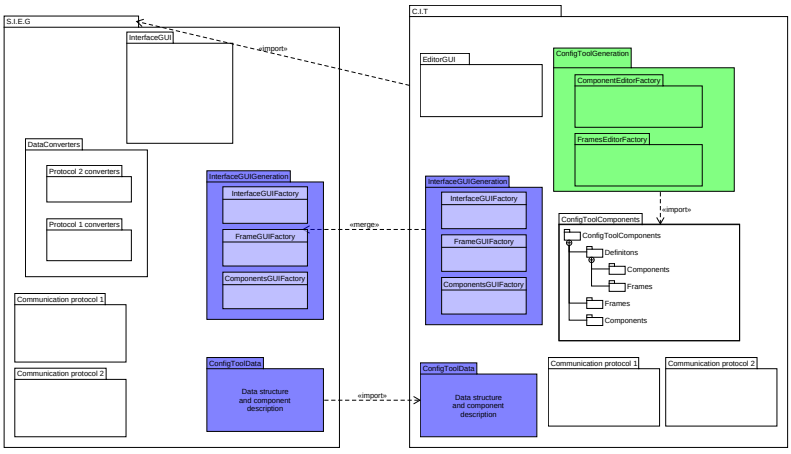


Figure 3.29: UML package diagram of the most important modules of S.I.E.G.Fr.I.E.D framework.

3.2.2.1 Data Structure Design

The data structure covers a central role in a data-centered architecture. In order to have a modular development both from the C.I.T and S.I.E.G this structure has to support dynamic collections, in which elements can be added without changing the overall structure. The encoding of the data can be binary or textual, in the first case many object-oriented languages offer the class serialization feature which is a way to convert code classes into binary file format. In the second case the use of a standard like XML is suggested and instead of serialization, parsers modules have to be implemented (also its inverse function has to be implemented to have both importing and exporting mechanisms). Both methods grant the realization of dynamic collections like generic type arrays, ordered sets, trees and maps. Device parameters, presentations data and every other feature are arranged in a vector of objects, paired with an array of descriptors. Every object contains a data structure specific for a certain function. For example there are data structures which contain the data mapping, frame composition, components that belongs to the frames, navigation trees and all the other functionalities provided by the framework. This process is recur-

sive, that means that a data object can contain a collection of other objects depending on the purposes. The fundamental part is that there is the definition of a generic object which can contain any data structure. Using a data format like XML this operation can be easily done because the interpretation of the elements is done through a parser which interprets the data according to the specification. In the case of object serialization which is the actually developed solution, this is possible by using the **reflection and inheritance** mechanisms. Inheritance in this case is implicit because every class that can be defined in Java language has the Object class as ancestor that means every user-defined class is itself an object class. Reflection is a mechanism of object-oriented programming (OOP), which basically consist in the possibility of specializing a generic object and understanding its class type through its instance, this is a simple explanation which does not define completely this mechanism, but it is useful to understand its use in this context. The instances of these classes are the specific device data-structures, that are stored in a generic container, which is the object, in order to use the same behaviors for any of them in the storage procedure. Through the associated descriptor it is then possible to understand which kind of instance is contained in every generic object and instantiate it as the specific class, this property is often called **structural reflection**. The use of these properties and a generic container allow a structure which is able to add other elements without knowing how they are composed. This strategy is effective in facing two scenarios which are common and hard to handle. One is development of modular application structures and the possibilities to extend modules and their features, through the addition of modules and elements inside them.

Every functionality which could be reused in the development of a software interface is implemented through a set of modules which rely on a single data item contained in the object vector. This simplifies the integration of new features inside the application options through the implementation of editor, generator and data modules. The extension of frameworks features and elements is the only part that requires new coding tasks, and as can be perceived by the above description can have two different scenarios: The addition of a feature which evolution is something improbable and modules that are intended to evolve and integrate new options and element. For

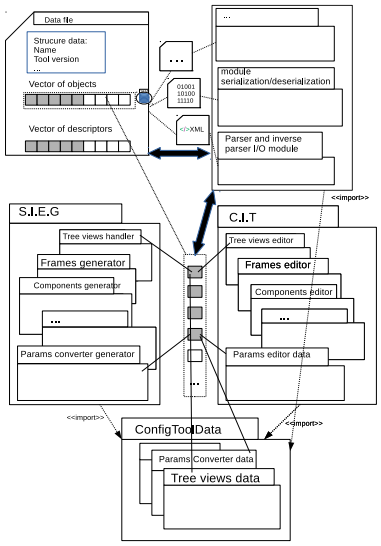


Figure 3.30: Composite diagram with UML package elements and custom model of the data structure and their decencies and interactions.

example the navigation feature based on tree structure despite its presentation is the same in quite every program ever done, and once the editor enables to represent any possible tree this module can be self-confined and never encounter modification, but for a module like the GUI elements generator it is quite sure that extending application ranges will have to renew the "look and feel" of the user interface, needing the development of new base components with different appearances. Through the use of such data structure the first scenario, see fig. 3.30, can be satisfied through a defined structural pattern that is the one proposed here that can be called Editor-Generator-Data-Modules (section 3.2.2.2). The second case, see fig. 3.31, which implies the extension of the elements of a module, always relies on similar mechanism that use dynamic data-structures plus the application of factory based patterns, through the extension of the set that can be generated through this pattern.

Figure 3.32, shows in more detailed manner the modular use of the data-structure highlighting the classes which are interested by the extension of application modules, showing the confinement of this process in two defined points (exporter and importer

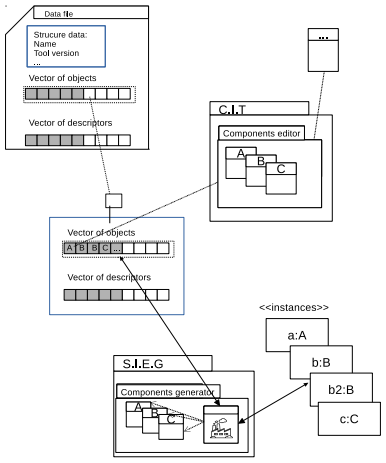


Figure 3.31: Composite diagram: UML packages, classes and custom model of generic module data structures.

class) underlining which makes the procedure guided, modular and flexible. To understand the modularity and the advantages related to the adoption of the factory pattern it is necessary to analyze some construction aspect of this method and will be discussed in the next section.

3.2.2.2 Architectural Pattern: Editor-Generator-Data-Modules

The architecture of the system, based on the design strategy introduced in the previous section, is based on this architectural pattern. This pattern, that has been thought and created for this specific context, is what permits to extend the application in a modular and guided way and granting that every added module can be reused in the automatic generation of new interfaces. This kind of scheme could be considered similar to a CAD for software development, which is a program which is used to build another one. CADs or IDEs are finished products that can be extended in their features through new releases or application plug-in and their design is something that

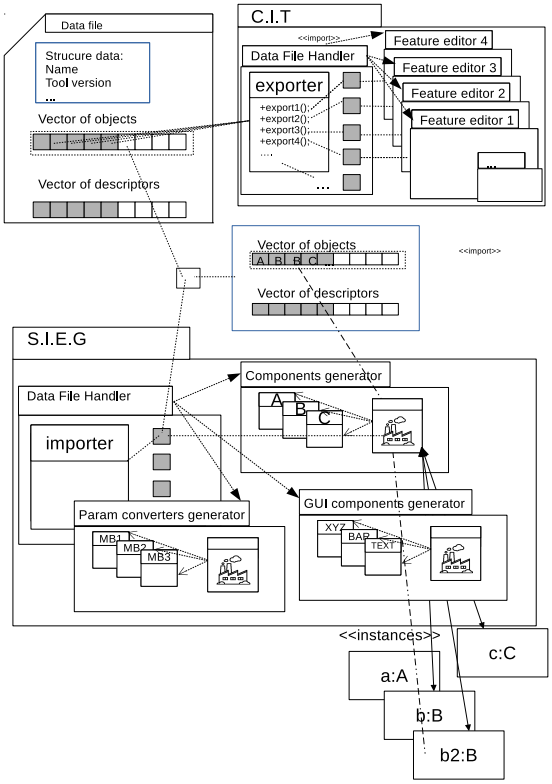


Figure 3.32: Composite diagram: UML packages, classes, objects, interactions with custom data structure model.

is not inherent in what will be result of the design that these tool are intended to aid. These kind of structures create a strong dependency between editor modules and the generated product. The design of the final product is nested in the design of its own editor. This concept binds the evolution of the two instruments. The binding of this evolution is what makes the system reliable and flexible at the same time through the definition of the infrastructure which guides the addition of new features through a defined process. This is possible because the product has the same nature of the editor which are both programs. The extension of the application features developing new modules has to follow a defined path. The process is composed of these activities:

1. Definition of a data object which is added to data module (ConfigToolData).
2. Reservation of a descriptor and an index in the data file.
3. Design of the editor module which can manipulate the data option of the new added feature.
4. Design of the interface generation module which can comprehend a GUI part or not.
5. The newly added modules are tested inside the C.I.T and when they are ready and documented the generation part of the new feature is merged inside the libraries of the S.I.E.G software.
6. The C.I.T file generation shall invoke also the methods of the new module when generating the data-file adding another object to the vector.
7. The S.I.E.G application shall invoke the generation of the new components using also the new object.

The addition of new elements inside modules is simpler because it implies only the insertion of the new element definition inside both editor and generation module and an extension of the factory class which shall be able to handle it, see fig. 3.31. In the development of modules it is important to follow the Model-View-Controller (MVC) pattern [58], physically separating the classes that compose a module in the three groups:

- Classes that represent and store the data of the module.
- Classes that implements the logic of the feature.
- Classes which compose the graphical interface and presentation layer of the application.

Not every module requires all these three groups, but this confinement can increase the modularity of the implementation especially if proper construction techniques to

reduce dependencies between them are used. In order to have a significant reduction of dependencies Publish-Subscriber pattern can be used to exchange data and actions between modules parts. This process may seem similar to an existing paradigm Construction by Configuration (C-b-C), and in fact uses the concepts of this approach but applied in a context which is not the one in which this approach has been matured:

"Construction-by-configuration (C-b-C) involves adopting systems, such as spreadsheets, that have different programming models than conventional application platforms and different processes are used when software is configured rather than programmed"[57]

"...this radical change in software development practice has been ignored by the software engineering research community. By and large, the concerns of that community remain those of the 20th century - methods and techniques for specifying, designing, programming and testing software that is written in a conventional programming language and whose source code is available to the software developers."[57]

The title of the article is "Software Construction by Configuration: Challenges for Software Engineering Research"[57] and for what concern my personal experience in the development of many software programs, I can say that both traditional construction and these methods are challenges for developers and researchers. These challenges are not only in the definition of new strategies and methods, but also in their practical application which often has not the necessary modeling languages and instruments to be documented, evaluated and taught, underlining how many aspects can be improved [59] [60]. Despite that in the next chapter and in all the writing will be presented, discussed and documented methods strictly inherent to construction aspects also with models which simplify their understanding especially in the realization of the C-b-C, even if the proposed paradigm is introduced fro the Enterprise Resource Planner (ERP) softwares and has a different target respect to the softwares that are presented here, it can also be used in other fields, like motor drive interface development, like has been done in this work, facing different application

environment.

This process and the structure on which it is based, constitute the pattern of the architecture of the system. It can be considered a design pattern due to the fact that this process is a common element in the extension of the software framework. The application of this pattern requires the presence of two design strategies that are the C-b-C approach, specialized in this application context to generate GUI elements and process variables handlers which translate protocols data into their digital image of them, and the data structure centered design. The data structure has also the requirements of being modular, as previously described, the size should not be fixed and the contained elements should not need specification of their composition to be added to the structure. These requirements are easily met by OOP programming mechanisms and available file encodings. Obviously the same design structure imposes constraints in the design of the modules which have to composed as described before, with a data structure, editor and generator module. This is a constraint, but also grants a decoupling in the data (the model) and the features logics, which is often a desired feature that is also the aim of some patterns, such as the mentioned MVC. All these requirements and constraints could seem an high price to pay, but they do not limit the possible features that can be added, they limit only their design, that has to be something at least similar to M-V-C, every added module can rely on the other already implemented or being totally independent, and eventual dependencies are contained in the corresponding module. This grants a highly modular and flexible structure, because there are no constraints on the number and nature of the added modules. The benefits in the adoption of this structure compensate the required efforts, it enables the mechanisms of editor and generation strategies maximizing the reuse of code through an effective use of C-b-C, avoiding the introduction of new lines of code to develop systems based on already implemented features, also increasing their robustness due the impossibility of adding wrong instruction sequences. The enabling of these possibilities is not the direct consequence of the architectural strategy, which is the the modular and extensible infrastructure, which can introduce new features and in that way follow the evolution of this trending environment which is the embedded devices world, in this case the industrial ones. These consequences are enabled by

the architectural strategy and realized also with the use of many design strategies.

3.2.2.3 Architecture Design

The architecture of the software system part which is composed of the two introduced programs, is based on the employment of already known and recognized design patterns: Mediator, Iterator, Observer, which are of the behavioral type, Factory and Prototype which are creational ones and some other maybe less known like Binding Properties [61]. The use and composition of these patterns in the design of the whole system is what makes possible to achieve all the described features of the software framework and all its automatisms. These patterns are applied in the implementation of the packages inherent to specific functional roles and their interactions. The iterator pattern is a design strategy for the case in which there is the need to explore a collection of elements and is often used with the data structures, for example Java and C++ libraries that implement the vector data structure have the iterator object, which has methods to iterate the inspection of the contained elements through apposite methods. This is very useful to automatically explore data structures inside conditioned "while" or "for" code loops in which the terminating condition is the reaching of the last element of the collection. A generic iterator pattern can be considered as the operation of exploring a complete set of elements and access them. Mediator is a strategy in which an object is responsible for the decoupling of entities which do not reference directly between them. Observer patterns provide a scheme in which there is an object which is the publisher and many possible subscribers which are registered to the publisher with a proper class interface implementation and is used to decouple objects in the case of one to many dependencies in which there is the need for propagating information and keep all of them updated. This is also the base of M-C-V and quite every GUI library has some provided implementation of the observer pattern to propagate user interfaces events and interactions to the objects that implement the application logic to separate the view (presentation) from the program logic. The factory method is a pattern which is used when there is the need of creating sets of similar objects in response to some application needs, in which the size and composition of the needed set is not known at the moment of the writing of the

program logic, or simply there is the need of a set of elements that can be created in more automated way. Binding Properties is a composition of multiple observers with the aim, as the name suggest, of keeping synced some properties of a set of objects and keep the state of that information updated and coherent through all the set.

The C.I.T and S.I.E.G structures have been already introduced through the UML package diagram with the representation of the most important modules, many of which belonging to both programs, in fact S.I.E.G is nested inside the other tool, mainly for the possibility of generating the final software interface program also through the editor in order to make easier the testing of new components without releasing them until they are ready and properly validated.

Once the new components are validated, the generation package containing the new generation module can be merged with the distributed version of the S.I.E.G, as suggested by the underlined relationship in the diagram, to be used in the development. S.I.E.G and C.I.T can also be organized as separated softwares that refer to the same modules, but this is a matter of application deployment preferences and projects folder organization to aid the process which generates executable code. The adopted and suggested approach is to treat the S.I.E.G software as a nested application which can be used also as a separate part that can be deployed as a standalone application. All these aspects are important but are not directly involved in the design, but in some constructions activities which are known art of the field and they are not relevant for the understanding of the results of what is described here.

The editor program (C.I.T) has a simpler design rather than the nested one, the main applied strategy is the mediator to interface the various element of the data structure with the proper modules. As intended by the overall guidelines every editor module is dedicated to a specific feature which refers to a data structure in a defined and indexed position inside the vector that contains it. Data handlers package of the C.I.T centralize the importing and exporting operations of the file that contains the data and assign its elements to every editor module updating its content. In a similar manner it also gather all the data from them in the generation of the file. This simplifies the file handling because every added module refers to an element of the data

structure, which are all handled by the same object. This makes modules independent from this process and they remain independent as they were thanks to the data design.

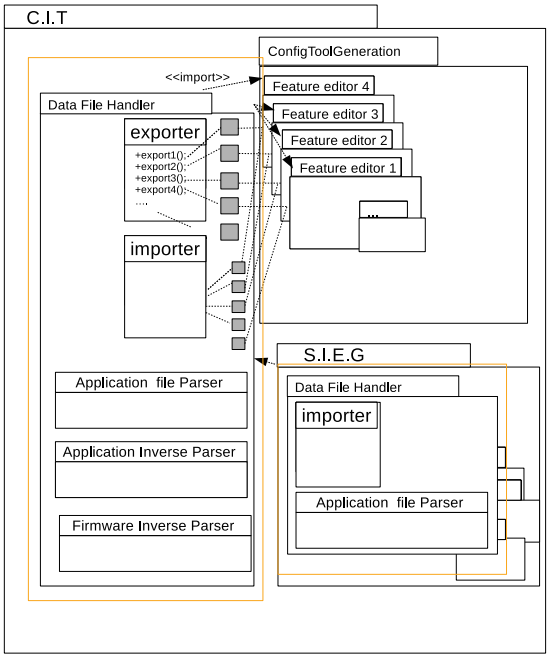


Figure 3.33: UML package diagram of C.I.T modules.

The file handler also perform the check on the imported data structure; suppose that an old configuration is imported after having added new modules, in that case the structure will lack the data elements that are inherent to the new features, and the corresponding module is not updated and works with default values. In the exporting of the file instead the corresponding indexes will be filled updating the whole structure. The S.I.E.G and C.I.T file handler are different because one handles the editor modules and the other one the generation modules. Editor module can rely on the generation ones to display what the editing operations produce on the selected feature. As already said every editor logic and presentation depends on the specific feature and its user interface is optimized to aid the end user in the usage of the particular characteristic. This approach also aids the configuration management, because

regardless the evolution of the instrument itself the missing data will simply result in the missing of a certain features, without compromising the whole functioning.

S.I.E.G design is complex, due to the fact that there are different activities which have many purposes, so there is the interaction between different strategies in order to handle them, in a unique organic structure that has to maintain flexibility and modularity. In the previous paragraph it has been already discussed how structure file is handled through the use of proper mediator implementation. Through the use case description it should be also clear what kinds of data are inside this structure, which are protocol maps and dictionary, GUI windows and components descriptions. The S.I.E.G file data handler mediator package has the purpose of distributing the imported data through all the system modules. As already said this kind of structure triggers the activation of generation module through the data presence. For example a device that supports only one communication protocol will have only the relative information and will generate only objects for the handling of that protocol.

The data handler package collects and implements Protocol Data Managers classes, every class uses the singleton pattern which ensures that inside the program there is only one object entity of this type of class because the handling of the protocol logic is totally managed through this entity. Every manager imports the corresponding data converter package. Converter packages are built with a mixed strategy of iterator and factory method patterns deputed to the generation of converter objects, one for each process variable that is shared through the system. The iterator explores the data inherent to the mapping process and passes the element to the factory, iterating upon all the set of information. The objects created by the factories derive all from a common parent class which has the purpose of exposing three methods that have the purposes to:

- convert process variable in protocol data and send through protocol mechanisms.
- update process variable through protocol data.
- call the update method of all GUI elements subscribed to this object when the process variable is updated.

As the pattern states the parent class can be an interface that is then implemented or a

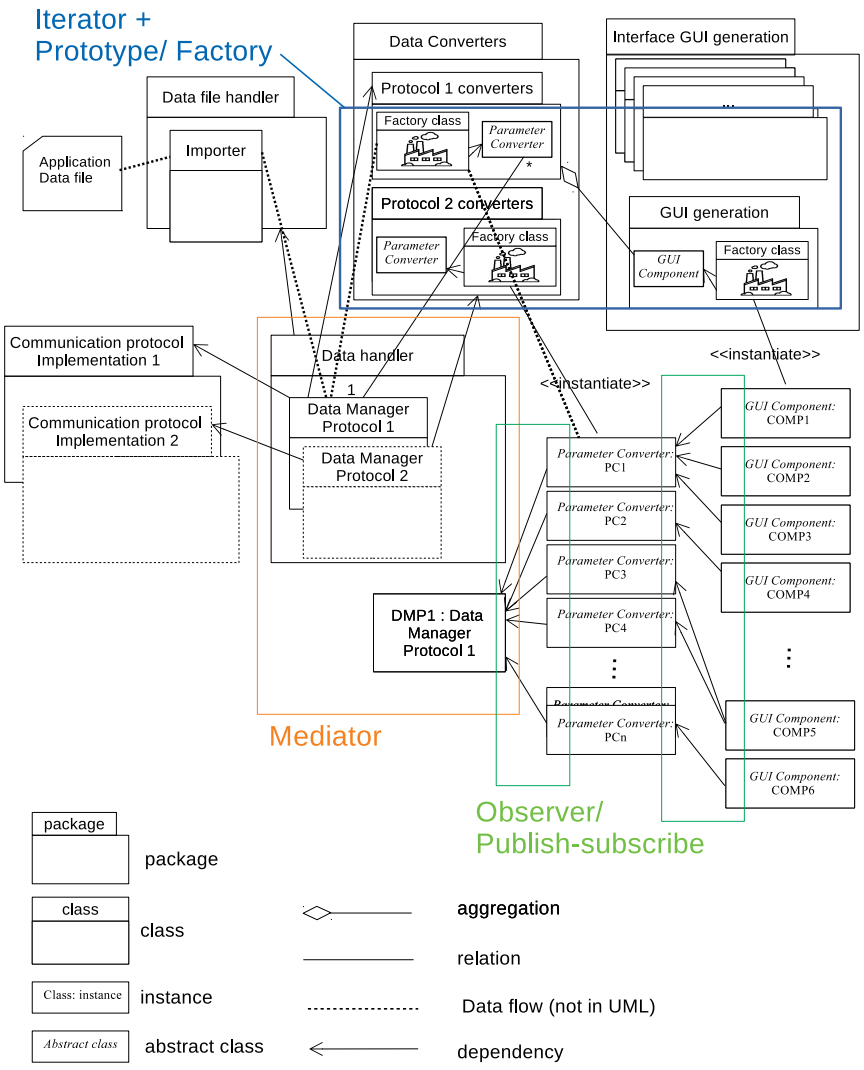


Figure 3.34: Composite diagram: S.I.E.G.Fr.I.E.D UML packages, classes, objects and patterns application.

class that is extended and the methods are overridden by children classes. This operation grants the realization of a set of variables converter which are specialized for the type of the process variable. Every derived class implements specific rules to convert protocol data into a variable, that depend on the variable encoding, that are the primitive data type, and the protocol data structure definition, which usually has read and write, read only, or other access features, but in the end they have always to compose a primitive data type and the relative access. An example will be shown in the following paragraphs to understand how these classes are defined in a factory and how this mechanism automates the whole protocol handling. When data converter objects are created they are organized in a proper collection or data structure in proper update groups through which the data manager is able to perform a more efficient update mechanism during the exchange of messages through the protocol implementation. The design of these interactions depends on the protocol characteristics. The important thing is to to define a proper infrastructure to update the interested variables when a message is received or sent, even one by one. Proper strategies can avoid the need of calling the update methods of every component or the subscription of a high number of subscribers, without losing the automatism of this process, but granting better performances. When the collection of converters has been created they are subscribed, individually or through sets of them, to the update of data changes, by the mediator and the generation factory. Both creation and subscription are done through factory method that means the reuse of the same tested mechanisms on all the objects which are granted to be updated by the act of subscription during their creation. After the generation of the data converters they all refer to variables, the collection of these variables is the digital image of the process. Every variable is then supposed to be manipulated by the proper GUI elements. GUI frames, views and their elements are generated with a similar mechanism always exploiting the iteration of the data structure and the factory processing. The properties of these elements are inserted in the data file when exported by the C.I.T and the same properties are used to generate them automatically. All the GUI elements have a parent class which provides at least a link for one process variable and accordingly to the data description of the component they are generated and subscribed to the update of the needed variables. The

combination of these observers make it possible to propagate data through the user input to the interested process variable, any change triggers the update of the interested GUI components connected, so if there are more representations of the same data they are automatically updated with the new edited value. In that sense they realize "Binding Properties" pattern, that means process variables and their update are bound together inside the whole system and these data are always coherent thanks to this propagation mechanism. The other generation modules till now implemented which are frames and navigation tree generation do not require complex interactions as the two previous one. Frames can be easily generated through a factory iterating their descriptors, they all have the same behavior and differ only in the layout properties which can be easily reconstructed. Navigation trees can be generated in the same way and every node contains the identifiers of linked frames which can be addressed in the generated set and make them visible or not according to the interaction with the navigation structure. In the Java implementation of these two last elements it is very easy to realize this kind of modular structure using the GUI library mechanisms that are based on the observer pattern to propagate interface actions and events between them. The combination of these different patterns, in the case of the realization of a "Binding Properties" behaviour with automatic generation of components and the automatic handling of protocol communication features, granting in both situations data coherence and propagation thanks to an effective implementation of C-b-C, can be considered new patterns themselves. They respond to the need of achieving this level of reuse of code and automatic handling in the realization of software with GUI interfaces with industrial protocol support for communication with embedded electronic devices.

Factory Propagation Pattern The binding of parents classes of factories, through an aggregation relationship, can generate a set of elements with an implicit subscription mechanisms defined in the creation of objects. The first factory provides elements which contain properties that are interesting for a subset of elements belonging to another set. The aggregation means that an element of the first set can contain one or more elements of the second set, see fig. 3.35. In this particular application the first set is the generated data converter layer in which a collection of objects is used to translate raw protocol data into the process variables, every process variable can be represented with different GUI elements inside different views. When GUI elements are created they are immediately bound to the variable they have to expose to user interactions. Variables objects register these components in an internal vector on which an update method is called every time the variable changes its value. The update method is specific for every child class of the GUI elements and presents the value of the variable according to its graphic representation. The GUI components factory, during the generation, accesses to the converter set and for every component takes a reference of the needed variables which is passed to the GUI element. GUI elements manipulate the variables through user input using read and write methods of the process variables which are parent class methods. The GUI element copies the reference of the instance of the process variable in a generic parent reference and when the read or write method is invoked to manipulate the variable the polymorphic characteristic of inheritance uses the child method implemented in the instance class, allowing to treat the variables as generic for the GUI component but executing the proper encoding function in read and write operations.

For example, a GUI component which give the textual representation of associated process variables, one of the float type and another one which is an integer 16 bit variable, have data that are different in memory occupation and also in the encoding, the same GUI component has to take as input a real number in the first case and an integer in the second and handle them in the same way but granting their correct encoding. This operation can be simply done through a different conversion of the String that the user writes in the box to compose the number. If process variables derive from a common interface class with read and write methods that operate on

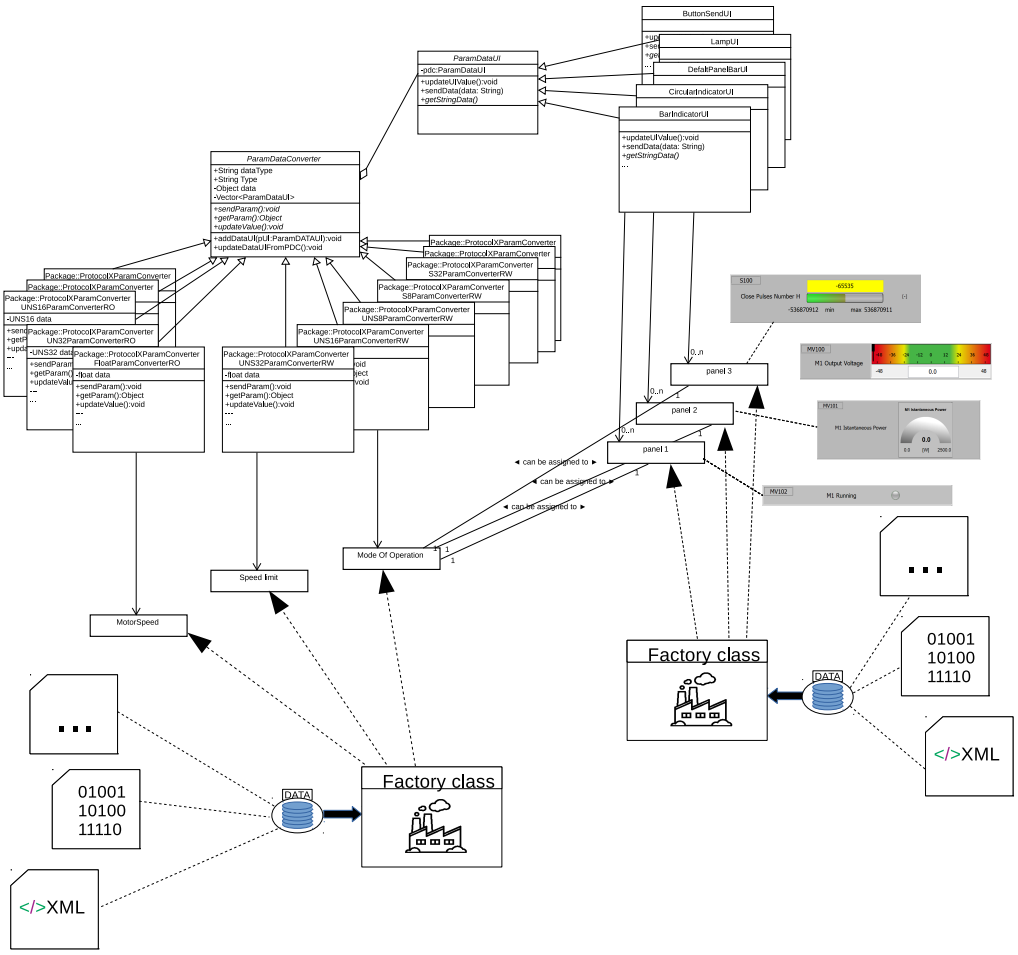
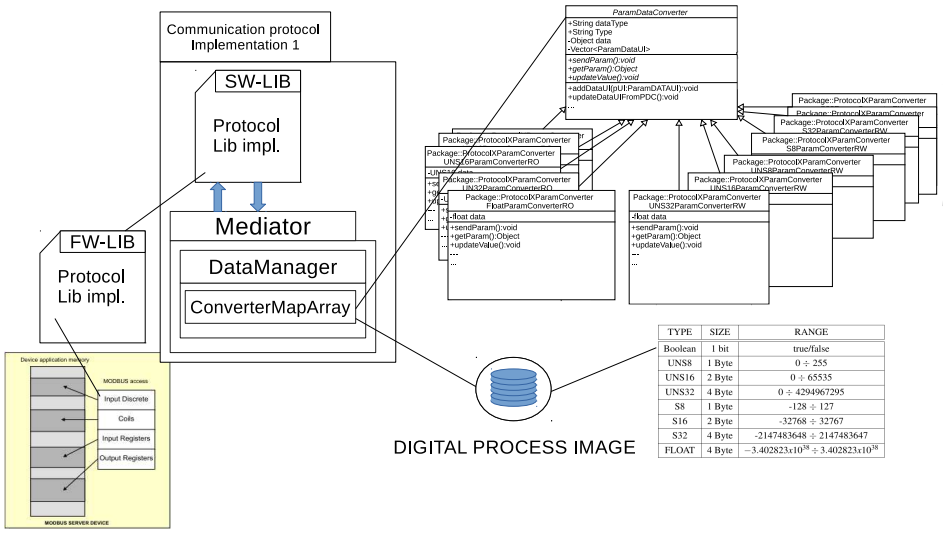


Figure 3.35: Composite diagram: UML classes, objects and relationships between them, with additional graphical exemplifications.

string values it is possible to derive a class for every data type and specialize read and write methods to convert the string in the proper format and vice-versa, which is an easy operation using Java String parse methods. When the GUI component manipulates the variable it uses a parent reference to the child instance, and calling read or write method will execute the corresponding method of the class which the instance belongs to. The generation of these elements is also automatically handled iterating the mechanism on the data structure which contains the data model of the various graphical elements and converters. A mediator presence is needed to handle the factories interaction for the registration of the elements of second set to the ones of the first group, which, as already said before, are stored in a proper data structure indexed through their identifiers, which is the information needed to establish the link between a GUI element and the corresponding process variable or variables. The mediator also acts as access point to the proper data structure that is forwarded to factories. These design strategy will be cleared in reading about its construction in section 3.2.3.

Data Converters Pattern is part of the core design of the system, the object converters are what makes possible to create the coherent digital image through the system, that is independent of protocol logic. Through the implementation of a proper mediator that contain the protocol implementation it is possible to define a set of classes which convert protocol data into process variables through their mapping information which are the parameters of the instances. The mediator is the entity deputed to handle the converter through the indexed container and also forward messages to and from converters. It has the purpose of decoupling application logic that access the process variables objects from the protocol logics and encoding rules, that handle the data transfers.



TYPE	SIZE	RANGE
Boolean	1 bit	true/false
UNS8	1 Byte	0 ÷ 255
UNS16	2 Byte	0 ÷ 65535
UNS32	4 Byte	0 ÷ 4294967295
S8	1 Byte	-128 ÷ 127
S16	2 Byte	-32768 ÷ 32767
S32	4 Byte	-2147483648 ÷ 2147483647
FLOAT	4 Byte	$-3.402823x10^{38} \div 3.402823x10^{38}$

Figure 3.36: Data Converters Pattern elements representations

The effectiveness of this strategy can be significantly increased using the previous pattern to automatically generate converters through the S.I.E.G.Fr.I.E.D approach. Changing the mediator and converter classes implementation it is possible to handle different protocol logics that refer to the same objects variables.

Data Converters Layers and Process Variables Data converters layers realized with the proposed pattern are able to handle automatically the communication protocol mechanisms to exchange data between devices and application only through the compilation of forms which aid the representation of the data maps (dictionary or tables). Dictionaries editors are often used to aid the firmware map generation of the protocol stack because they simplify the handling and the following of the evolution of these structures. In this case the editor generates both the firmware code and the corresponding software map. The already introduced benefit is that the two data structure are synced and also their evolution, and in addition to the traditional editors it also generates a complete commissioning software, using the same editing process, with the exception of the graphical options that are a small price to pay for the produced output. All data converter layers refer to the same collection of process variables, which are contained in a proper indexed data structure. Every process variable has an unique id assigned during the editing operation that is used to identify them. The identifier is also shown by every graphical component which makes easier to explore the documentation in which can be proposed the same identifier that is linked to the explanation of the purpose of the variable.

The indexed data structure is also very useful because it contains the digital image of the device represented through numbers which are directly understandable as physical quantities or some characters representation. If the containing structure can grant the dynamic behaviour as the one used for the application it can be extended easily in case new variables are added. Using object serialization or other encodings it is possible to export and import this data structure as it happens for the application file.

That means that devices configuration and digital images of the process can be exported and imported through the application, see fig. 3.37. Importing the configu-

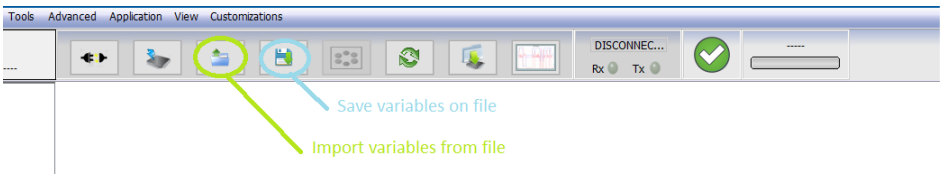


Figure 3.37: Generated application tool-bar with import and export options.

ration can update the variables which can be then sent to devices through the proper mediator depending on the used protocol. So it is possible also to share the same image between devices that support different protocol mechanisms and treat them indifferently. Exploiting the same concepts of the modular and dynamic application data structure, the import and export operation can benefit of the same trigger mechanism. For example suppose to export a data configuration made with a certain version of the application in which is expected the presence of a certain set of process variables.

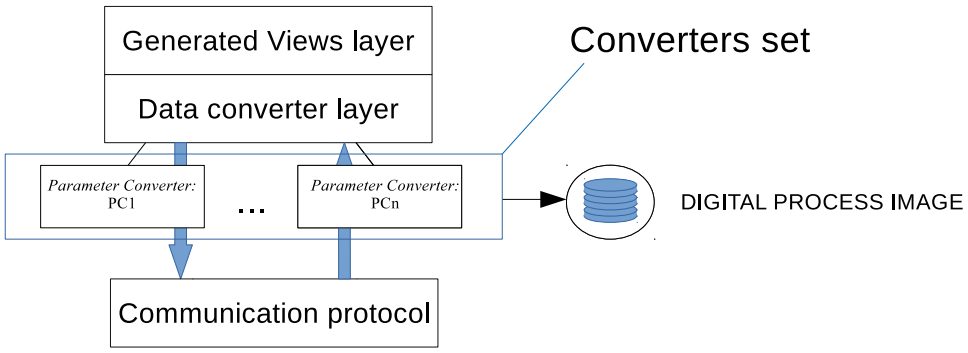


Figure 3.38: Data converter layer with converters set referring to process variables.

Then a new release is made with the addition of some variables creating a different configuration set for the device, then when importing the old data structure the

missing variables in the imported file will be left to their default values, while the present ones can be imported to update the corresponding digital image. The same concept works in the case the variables are removed (for example debug variables can be removed from a version to another); in this case the imported variables which indexes are no more present in the configuration will be simply discarded.

Protocol Converters Synchronization The actual implementation is supposed to use only one communication protocol at once and to transfer data between devices that use different protocols it is necessary to export and import the digital image and then transfer it through the proper converter layer. In this scenario it is not excluded the possibility of communicating with the device through different channels and then realize a synchronization between two or more converter layers in order to create a coherent process image shared between them. At the moment this scenario has not yet implemented even if is contemplated and at the moment the idea is to use the observers and mediator to propagate changes in the process variables to the internal protocol structures of the data. The infrastructure itself enables the possibility of building such mechanisms to synchronize different communication channels, at the moment no real application scenarios with this need have been encountered but this feature will be certainly developed in the future and possible applications will be evaluated. Generated data converter layers collect objects that refer to process variables and aid their manipulation through simple read and write methods and their construction which separates them from the view layer. It also gives the possibility to use this layer as a separate API to access the digital image of the device and to build applications on it without caring about the communication handling. This is very useful for the prototyping of supervisor application logics, and to realize particular customizations of the application. Due to the reduction of time that is granted, a concrete prototype of a certain functioning can be realized only focusing efforts on the application logic and also can reuse every element of the framework libraries. Then if the new applications are interesting their logic can be also implemented inside the editor environment and a new feature can be created.

3.2.3 Software Construction

When dealing with the effective implementation of the design strategies, their construction is a critical point because new constraints and options are encountered in this stage. The construction of the program is the activity which realizes it, through the writing of program files and their manipulation to obtain the executable application. Languages and tools introduce specific constraints and capabilities but do not offer any support in the implementation of the planned strategies, because the freedom degrees are uncountable. This is one of the points of strength of the virtual elements, which have the intent of being extremely flexible and their technologies allow it. With this universe of possible paths with specific constraints, implementing desired strategies can be very difficult and it is quite impossible to have a unique optimal solution, unlike what happens in engineering problems that can be solved with mathematical models in closed form. Implementing an algorithm with a logical or mathematical base is something that can have a unique optimal solution, even considering the computational and programming constraints, but the design of behaviors and architectures is something that requires a high abstraction process because the logics behind them have some differences from a traditional mathematical model. This abstraction process requires a lot of creativity, but also an extremely rigorous approach due to the logic that every piece of code has to fulfill. In an article of 1996, which I think describes the same problem that is still open at the present time, entitled *"Who cares about software construction?"*[62], is explained a concept that is still addressed today, and that has been, and is still, an open issue.

"Researchers, consultants, and writers who dismiss construction as a trivial activity need to realize that construction, by definition, occupies the central role in software development. It is where the rubber meets the road, and that makes it a uniquely productive area in which to focus our attention. " [62]

Software construction, due to its practical nature and its complexity, has always been a difficult task, which has not drawn the necessary attention, unlike software design, which is very discussed and there are lots of literature examples about, such

as the previously cited book "Design Patterns".

"In May 2001, trial version of the guide to the Software Engineering Body of Knowledge (SWEBOK) was released in a Web format and, in December 2001, in book format with the intent to collect comments and possible improvements. Up to now, feedback received confirmed the usefulness of the guide for all documented knowledge areas, ***with the exception of the software construction knowledge area, for which the content does not map easily to industry practices or to actual academic curricula.****"*[63]

There are many cases in which a lack of attention and efforts investment in the study of this stage is reported, with the result that there are few attempts in facing this issue and often with an excessively generic approach which does not suit this context that is extremely specific and is often aside of trending subjects, like model based design and framework approaches, which do not focus primarily on the construction subject.

"The construction of modern software projects with a multitude of quality requirements is a challenging and complex task particularly for large scale systems. Underestimating this complexity may result in project failures at worst or the delivery of poor quality products that are plagued with software defects at best [64]."

This stage is not only an open issue but according to considerations taken from known literature and in the field experience, is the core activity in the realization of a software product.

"The irony in this shift in focus is that construction is the only development activity that is guaranteed to be done. Right or wrong, you could assume requirements rather than analyzing them, you can shortchange architecture design, and you can abbreviate or skip system testing. But no matter how rushed or poorly planned your

project is, you cannot skip construction. If there's going to be a program, there must be construction. Because construction is the only activity that must be done, code is often the only accurate description of the software available, which makes it imperative that the source code be of the highest possible quality ... Such detailed techniques must be applied as the code is constructed - it is virtually impossible to retrofit the thousands of picky details that spell the difference between a maintainable program and a failure. "[62]

This last sentence, always from the 1996 paper "who cares about construction?" explains my opinion on the subject better than I could do. This consideration is valid for software but in the chapter regarding firmware development it will be shown that in such environment this issue is even more much resented. Software construction techniques and methods are not only an open issue, they are the part of the software development process which has an high impact on it. Proposing solutions to this kind of issues, is not only a push to innovation but also a significant contribute to the core part of a process and this should make clear its importance. Software and Firmware are in continuous evolution so is their construction, that is why in my and others opinion that is a unique productive area which should be properly explored and maintained in order to follow this evolution. The next sections present some of the techniques used inside the development of the framework patterns and instruments, underlining the advantages that they can grant. Construction is supported by guidelines and design pattern but the proper application of these strategies, but the effective implementation of the advantages that they can achieve has to be supported by the proper construction.

The IEEE SWEBOK, in its description of the body of the knowledge of software engineering, in the chapter inherent to "SOFTWARE ENGINEERING MODELS AND METHODS" states that:

"Methods provide an approach to the systematic specification, design, construction, test, and verification of the end-item software and associated work products."[16]

The main topics of this knowledge area are:

- Modeling.
- Types of Models.
- Analysis of Models.
- Software Engineering Methods.

Always in the same chapter, concerning methods:

"Software engineering methods provide an organized and systematic approach to developing software for a target computer. There are numerous methods from which to choose, and it is important for the software engineer to choose an appropriate method or methods for the software development task at hand; this choice can have a dramatic effect on the success of the software project. Use of these software engineering methods coupled with people of the right skill set and tools enable the software engineers to visualize the details of the software and ultimately transform the representation into a working set of code and data."[16]

As should be clear also from this work, when dealing with virtual products, the modeling activities are done to have a more concrete idea of what they are and how they work. It is a fundamental activity in this branch of the engineering. Models are also used to validate the quality of the product itself [65]. Methods build the process that starts from the models, that represent the software functioning and structure, ending in the implementation and deployment of the programs. According to this book, methods have four classifications:

- Heuristic Methods, as the name suggest, are based on experience and widely practiced in the software industry.
- Formal Methods, which are based on rigorous mathematically notations and languages.

- Prototyping Methods, that is an activity that generally creates incomplete or minimally functional versions of a software application, usually for trying out specific new features, especially when new features require innovative approaches.

- Agile Methods, which are considered lightweight methods characterized by short and iterative development cycles, self-organizing teams, simpler designs, code refactoring, test-driven development, frequent customer involvement, and an emphasis on creating a demonstrable working product with each development cycle.

Regarding the development of software with OOP paradigms, this is considered an heuristic method, in fact the modeling activity and design do not follow a rigorous mathematical notation but an approach that is based on the "object concept", that relate programming aspects to an higher abstraction similar to domain logics [66]. The book definition is the following:

Object-Oriented Design Analysis and Design Methods: The object-oriented model is represented as a collection of objects that encapsulate data and relationships and interact with other objects through methods. Objects may be real-world items or virtual items. The software model is constructed using diagrams to constitute selected views of the software. Progressive refinement of the software models leads to a detailed design. The detailed design is then either evolved through successive iterations or transformed (using some mechanism) into the implementation view of the model, where the code and packaging approach for eventual software product release and deployment is expressed.[16]

The Object-Oriented Design has matured very effective modeling instruments, such as UML, and the design methodologies have drawn a lot of interest, producing a rich state of the art, like some of the design pattern that have been shown in the previous section with the aid these diagrams. Even if not based on purely formal methods, this solution has been widely adopted, and probably is because this particular abstrac-

tion is very effective in the transmission of the concepts and structures of a program in easily understandable sketches. The construction of a software which is referred as the mechanisms that refine models into packages, files and line of codes, is something that has to be explored [67]. **"The detailed design is then either evolved through successive iteration or transformed *(using some mechanism)* into the implementation view of the model"**
Construction details are often given in the design through code examples of patterns and applications fundamental blocks. The detail level is often left to the expertise of the software engineers, which mainly have to make this stage comprehensible for the programmers in charge of the writing of the code. Always from IEEE SWEBOK Guide:

"The term "software construction" refers to the detailed creation of working, meaningful software through a combination of coding, verification, unit testing, integration testing, and debugging. There are few standards on the details of software coding. It has been found through (mostly bad) experience that coding conventions are not appropriate for standardization because, in most cases, the real benefit comes from the consistency of applying an arbitrary convention rather than the convention itself. So, although coding conventions are a good idea, it is generally left to the organization or the project to develop such a standard."[16]

UML and code examples have been proven to be valid approaches in development of complex software, and this software framework design could be a valid result to claim the above sentences. The major issues of this stage are the correct translation of models into code, as someone said *"is where the rubber meets the road"* [62]. The choice of *"the right skill set and tools"* constrains the Object-Oriented model into the language mechanisms, reason why, code examples are very important in completing the patterns description. A strategy that is not correctly implemented by the adopted tools will not achieve the purpose that is expected. These informations are also at the base of software documentation, which is the description of its composition, using style conventions and rules proper for the specific languages.

The boundary in the details of these descriptions, is when, writing the construction concepts, huge portions of the program code are written inside the documenta-

tion. This boundary has not an exact definition and depend also whom it is addressed to, final users or team developers. The elements of the construction process, that are important to document, are the construction mechanisms that implement the newly developed strategies, with the adopted tools, in the facing of new problems, and in the extension of what is already known art. If this process in done correctly, every time that a design concept is explained and implemented, with a specific tool, its guidelines, in both design and construction, can be used as reference for these stages and also for documentation purposes. This should be recommended way to avoid the resolving of previously faced problems. For the above reason and to focus the attention on the innovative content of this work, the following sections present construction examples and documentation regarding the most significant framework mechanisms. These sections are intended to complete the description of the presented design improvements. In my opinion, the weak point of coding standards and other conventions applied to the construction of a program, is that these rules are often thought with the aim of being general. Due to this fact, they are often applied for all construction contexts trying to benefit of the advantages that they should ward. Trying to generalize the construction of different projects through conventions is not always possible. Always my personal opinion is that conventions should be used, if they do not impose constraints on the possibilities offered by instruments. Sometimes conventions can not be in tune with the realization of some strategies. Always in my opinion, the only way to produce reliable softwares and quality projects, is to apply conventions when they can achieve their purposes without constraining the possibilities offered by the adopted tools. If there are no conventions suitable for the project it has to be designed a proper one. Then has to be proven that it is reliable for that solution, that consider both strategy design and construction instruments, with the aim of exploiting all the features that they can offer. Developing conventions and methods for specific implementations, means find a specific solution for a specific context [67], always citing an article that I find inspiring, is exactly "where the rubber meets the road" [62].

3.2.3.1 Tools and Instruments

The implementation of the software framework is done, as already said, with Java language and supported by the ECLIPSE IDE [68]. This IDE, is a very versatile tool, which has support for many of the major languages used in programming environments: Ada, ABAP, C, C++, C#, COBOL, D, Fortran, Haskell, JavaScript, Julia, Lasso, Lua, NATURAL, Perl, PHP, Prolog, Python, R, Ruby (including Ruby on Rails framework), Rust, Scala, Clojure, Groovy, Scheme, and Erlang. It can also be used to develop documents with LaTeX (via a TeXlipse plug-in) and packages for the software Mathematica [69]. It is free and open-source. Thanks to its structure, which has a very interesting design that can easily add plug-ins to develop custom editors, it has been widely used as base for many development environments, even in embedded micro-controllers programming, like CodeWarrior from Freescale (now NXP), Kinets Design Studio, MCU-Expresso-IDE [47] both from NXP, System Workbench for STM32 [70] from ST and many others. Java language has its own style convention, which is recommended to be followed in order to write "readable" code, through the style in which they are written. Eclipse IDE with Java environment offers many tools to create a rich development platform also with some automatic features, as UML class diagram generation from classes files, using ObjectAid plug-in, and many other tools for modeling like UMLET or Papyrus. The rich set of Java conventions and design examples are enough to have hints on how to document and write applications through consolidated construction processes. All this work is based on that solid background, which has been built on UML diagrams, Patterns, Java examples and their implementations of them. So the only innovative content in the construction are the mechanisms which enable the new proposed design strategies, which salient points will be shown below. There are some concepts that are the core of the construction of this approach, related to the automatic handling of data through communication protocols and their presentation generation.

3.2.3.2 Patterns Implementations

Constructions of known patterns, is something that is quite common in programming environments, like GUI libraries which often implement observer mechanisms, like the java swing one, through listeners classes. These are examples of construction of known patterns.

Observer implementation using listeners: listeners are a well known and effective implementation of this pattern, granting an useful decoupling between objects and data propagation, which makes it very suitable also for event driven strategies. It is possible to define this kind of structure for every object that needs it by the definition of:

- Definition of a subscriber list of interfaces inside the class that produces useful information; Methods for registration and removal of interfaces in the list. Every time there is new information it is propagated by an internal method which addresses all the registered listeners, see fig. 3.39.

```
// subscribers
EventListenerList myEventListeners = new EventListenerList();

        // subscribe
       public void addMyEventlistener(MyEventlistener myEventlistener)
       {
              myEventListeners.add(MyEventlistener.class, myEventlistener);
       }

       // remove subscribtion
       public void removeMyEventlistener(MyEventlistener myEventlistener)
       {
              myEventListeners.remove(MyEventlistener.class, myEventlistener);
       }

       // publish
       protected void fireMyEvent(MyEvent myEvent)
       {
            Object[] listeners = SerialMessageWriteListeners.getListenerList();
            // loop through each listener and pass on the event if needed
            int numListeners = listeners.length;
            for (int i = 0; i<numListeners; i+=2)
            {
                 if (listeners[i]== MyEventlistener.class)
                 {
                      // pass the event to the listeners event dispatch method
                      ((MyEventlistener)listeners[i+1]).dispatchMyEvent(myEvent);
                 }
            }
       }
```

Figure 3.39: Java code example of publisher construction for observer pattern implementation

- Definition of an event class that contain the useful data, see fig. 3.40.

```
public class MyEvent extends AWTEvent
{
            private static final long serialVersionUID = 1L;
            private Object myPrivateObject;

            public SerialMessageWriteToEvent(Object myObjectParam)
            {
                  super(source, id);
                  myPrivateObject = myObjectParam;
            }
}
```

Figure 3.40: Java code example of event construction for observer pattern implementation

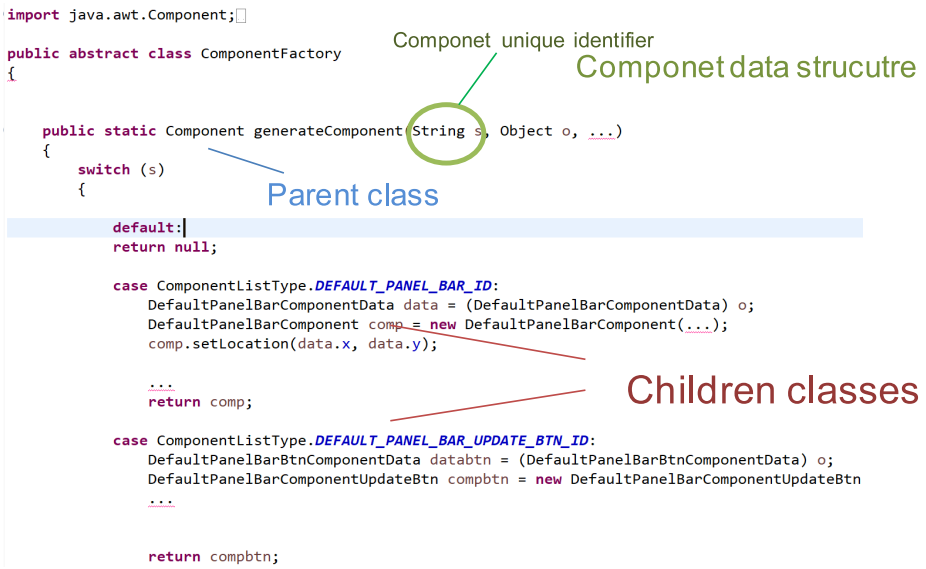
- Definition of an interface class which can be implemented in other classes defining a proper behaviour inside the inherited method, see fig. 3.41.

```
      /**
       *listener interface for MyEvent event
       */
      public interface MyEventListener extends EventListener
      {
            // event dispatch method/s
            /*
             * @param myEvent the event containing myPrivateObject
             */
            void dispatchMyEvent(MyEvent myEvent);
      }
```

Figure 3.41: Java code example of listener interface construction for observer pattern implementation

Factory Method Pattern: Factory method pattern implementations can be constructed in some variants. **Parameterized Factory Method** [49], is the one that has been used, and its behaviour is achieved passing data structures to the method, which act as parameters to choose the element that has to be generated. Data structure does not only contain the parameter that suggest the object that has to be created, but also some properties that will be used to create it. There are many possible ways to compose such behaviour, construction is a process that can be continuously refined. For example in the figure 3.42 below, the object parameter is the string that is associated to the object and these string definitions are stored inside a class instance (componentListType) which has static fields containing these Strings. These are compared to the field contained in the data structure. This could be also done with reflection

mechanisms, in which the data structure is specific for every component and derived from a common parent class. The class type that can be obtained, through run-time reflection could be used as parameter for selection of the object that has to be created.




```
import java.awt.Component;

public abstract class ComponentFactory
{

    public static Component generateComponent(String s, Object o, ...)
    {
        switch (s)
        {
            default:
            return null;

            case ComponentListType.DEFAULT_PANEL_BAR_ID:
                DefaultPanelBarComponentData data = (DefaultPanelBarComponentData) o;
                DefaultPanelBarComponent comp = new DefaultPanelBarComponent(...);
                comp.setLocation(data.x, data.y);

                ...
                return comp;

            case ComponentListType.DEFAULT_PANEL_BAR_UPDATE_BTN_ID:
                DefaultPanelBarBtnComponentData databtn = (DefaultPanelBarBtnComponentData) o;
                DefaultPanelBarComponentUpdateBtn compbtn = new DefaultPanelBarComponentUpdateBtn
                ...


                return compbtn;
```

Figure 3.42: Java example of parametrized factory method implementation.

In this case data structures are provided by C.I.T editor and this pattern, when combined to the iterator one, can generate automatically the objects that compose the application.

3.2.3.3 Automatic Parameter Converters and Presenter Pattern

This pattern is one of the core part of the software framework and is what enables the most interesting features of this work. The "Automatic" word in its name has to be supported by proper mediator and iterator patterns, see 3.34 , that are used to feed the factories, see 3.35 in order to create the automatic iterative behaviour that converts data structures in objects. Supposing that the construction of a module, that is used to import and handle a data file, is a common process known by almost every programmer, this part will not be exemplified. Supposing also, that data structure, supports the characteristics described in the section 3.2.2.1. The operations that handle the file and iterate the data structure, are not different from the loading of an array in memory and its exploration in a conditioned code loop. Extracting elements and passing them to the method of a factory should also be a well known construction scenario. What instead has to be described is the construction of the mechanisms introduced in this solution. The elements involved in this pattern are:

- Process Variables objects, in which every variable derives from a parent class which has a string representation of its quantity, then the derived classes represent and contain the primitive data-types and the corresponding String presentation, see fig. 3.43 and fig. 3.44.
- Data converters, which are created one for each process variable and they are updated by the protocol mediator, with a proper strategy, that depends on protocol features, in which every shared message triggers the update of the interested converters.
- GUI components, that are created to interact with at least one process variable, which is manipulated through the string object.

As stated before, in this solution, Process Variables are the center of the whole mechanism, and they also act as interface between presentation and protocol data conversion, through the string representation and the primitive data-types.

The highlighted methods are used during the factory creation process, in which every converter has to handle a process variable, which is passed as function param-

```
package patterns.examples;

import java.util.Vector;

public abstract class ProcessVariable
{
private String variable = " ";
private String variableID = " ";
ParamDataConverter pdc ;

private Vector<ParamDataGUI> guiPresentations;

/* PRIMITIVE DATA CONVERSION TO IMPLEMENT */

public abstract void writeVar(String s);

public abstract String getVar();


public String getStringRepresentation()
{
   return variable;
};

public String getID()
{
  return variableID;
};

public ProcessVariable(String ID) throws Exception
{
if(ID.equals("") || ID == null )
{
Exception e = new Exception("NO ID");
throw e;
}
      variableID = ID;
      variable = new String("");
      guiPresentations = new Vector<ParamDataGUI>();
}

  public void registerPdc(ParamDataConverter pdc )
  {
      this.pdc = pdc;
  };

public void registerGuiPresentation(ParamDataGUI gui)
{
      guiPresentations.addElement(gui);
}

public void update()
{
      for(int i=0; i< guiPresentations.size(); i++)
      {
            guiPresentations.get(i).update(variable);
      }
};
}
```

Figure 3.43: Java code example of process variable class definition.

eter. Then the converter registers himself in the constructor method of the process variable, passing its reference to the "registerPdc" method.

```java
package patterns.examples;

public class ProcessVariableFloat extends
ProcessVariable
{
	Float f =1.0f;


public ProcessVariableFloat(String ID)
throws Exception {
		super(ID);
	}

	@Override
	public void writeVar(String s) {
		f = Float.parseFloat(s);
		super.pdc.send();
	}

	@Override
	public String getVar() {
		String s = Float.toString(f);
		return s;
	}

}
```

```java
package patterns.examples;

public class ProcessVariableS16 extends
ProcessVariable
{
	Short sh =1;

public ProcessVariableS16(String ID) throws
Exception {
		super(ID);
	}

	@Override
	public void writeVar(String s) {
		sh = Short.parseShort(s);
		super.pdc.send();
	}

	@Override
	public String getVar() {

		String s = Short.toString(sh);
		return s;
	}
}
```

Figure 3.44: Java code example of derived process variables classes definitions.

The method "writeVar", is the one that has to be used by GUI components to propagate user input. The "getVar" method, has to be used to give the feedback to the user, through proper graphical render of the String value or in the way that is expected to be presented by the component. The derived variables which are type specific have to parse the string representation into the proper type and during the "writeVar" operation they have to call the send method of the associated converter.

```java
package patterns.examples;

public abstract class ParamDataConverter
{
ProcessVariable prcVar = null;
ProtocolImplementation protocolReference =null;
ProtocolConverteData data = null;

public ParamDataConverter(ProcessVariable
prcVar,ProtocolImplementation protocolref,
ProtocolConverteData data) {
   this.prcVar =prcVar;
   this.protocolReference = protocolref;
   this.data = data;
   this.prcVar.registerPdc(this);
}

 public abstract void send();

 public abstract void update();
}
```

Figure 3.45: Java code example of Data Converter class definition

The invocation of the registerPdc method of the process variable inside the converter constructor is the operation through which these two components are automatically bound to implement the depicted propagation of information.

```
package patterns.examples;

public class RWS16PDC extends ParamDataConverter{


ProcessVariableS16 processvar = null;


public RWS16PDC(ProcessVariable var, ProtocolImplementation protocolref,
ProtocolConverteData data)
{
    super(var,protocolref,data);
    processvar = (ProcessVariableS16) super.prcVar;
}


@Override
public void send()
{
   byte shortbytes[] = new byte[4];

   shortbytes[0] = (byte)(processvar.sh & (0x00FF));
   shortbytes[1] = (byte)(processvar.sh & (0xFF00));;

   System.out.print("Sending bytes of short data 1 .. 2 .. ");
   protocolReference.sendByte(shortbytes);
}

@Override
public void update()
{
   System.out.print("Updating bytes of short data 1 .. 2 .. ");
   processvar.sh = (short) ((protocolReference.rwData.get(data.index)
   +protocolReference.rwData.get(data.index +1) << 8));

   processvar.update();
}

}
```

Figure 3.46: Java code example of derived Data Converter class definition for signed integer variables of sixteen bits size.

```
package patterns.examples;

public class RWFloatPDC extends ParamDataConverter
{
ProcessVariableFloat processvar = null;


public RWFloatPDC(ProcessVariable var, ProtocolImplementation protocolref,
ProtocolConverteData data)
{
  super(var,protocolref,data);
  processvar = (ProcessVariableFloat) super.prcVar;

}

@Override
public void send()
{
    byte floatbytes[] = new byte[4];
    int floatcovnerted = Float.floatToRawIntBits(processvar.f);
    floatbytes[0] = (byte)(floatcovnerted & (0x000000FF));
    floatbytes[1] = (byte)(floatcovnerted & (0x0000FF00));;
    floatbytes[2] = (byte)(floatcovnerted & (0x00FF0000));;
    floatbytes[3] = (byte)(floatcovnerted & (0xFF00000));


    System.out.print("Sending bytes of float data 1 .. 2 .. 3 .. 4 ..");
    protocolReference.sendByte(floatbytes);
};

@Override
public void update()
{
     int floatcovnerted =0;
     floatcovnerted = protocolReference.rwData.get(data.index) +
     protocolReference.rwData.get(data.index +1) << 8 +
     protocolReference.rwData.get(data.index +2) << 16 +
     protocolReference.rwData.get(data.index +3) << 24;

     System.out.print("updating bytes of float data 1 .. 2 .. 3 .. 4 ..");
     processvar.f = Float.intBitsToFloat(floatcovnerted);

     processvar.update();
}
}
```

Figure 3.47: Java code example of derived Data Converter class definition for float variables.

The derived classes are specialized to handle primitive data types through specific protocol implementation. Usually protocol raw data is organized in groups depending on the access type; like read and write, read only, write only and so on. Consequently the derived classes have to implement type and access, which can be easily done, creating different classes in which "send" and "update" methods behave as supposed, see fig.3.46 and fig. 3.47. For example read only integer variables, of a certain bit size, will have an associated converter class with an empty send method.

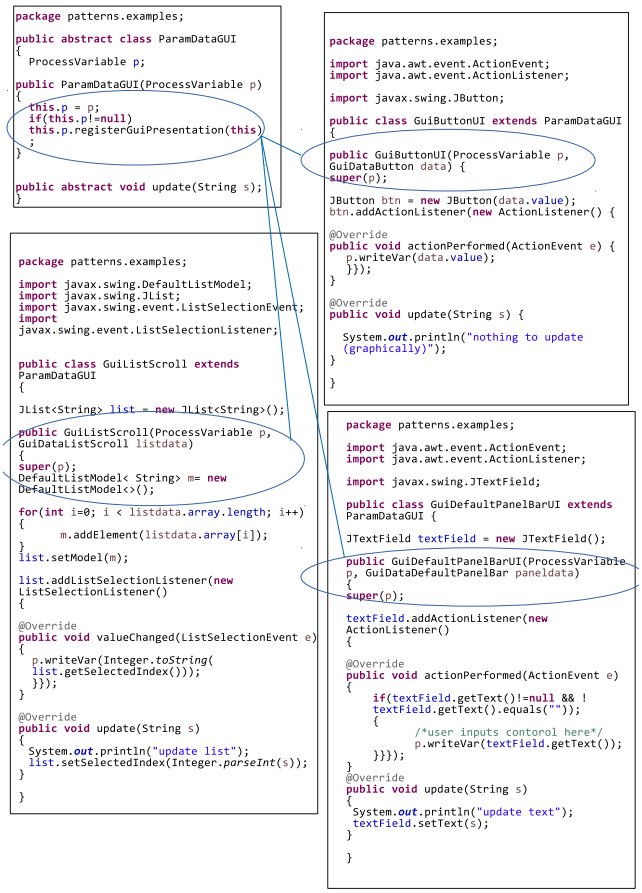
```
try
{
   //generation process
  s16 = new ProcessVariableS16("D2");
  f = new ProcessVariableFloat("D1");
} catch (Exception e) {

   e.printStackTrace();
}
RWFloatPDC floatpdc = new RWFloatPDC(f, modbuds, fdataD1);
RWS16PDC s16pdc = new RWS16PDC(s16, modbuds, s16dataD2);
```

Figure 3.48: Java code example, process variables and converters declaration.

The exemplified variables and converters generation, see fig. 3.48, can be easily automated through the parametrized factory pattern.

GUI generation: The same concepts are applied to the generation of GUI elements, which require at least a process variable at the moment of their creation. They register themselves through the proper method inside their constructor, see fig 3.49. All the necessary information is provided inside specific data structures which can be organized in the same manner of the components, using a hierarchical implementation of data structures which are specialized for GUI components. Every data structure contains specific information about the properties of the generated objects. These properties are the ones that are composed through the editor program. Combining inheritance, polymorphism, dynamic casting, reflection and other techniques of the OOP, these concepts can be easily implemented and they could be further improved. The exemplification is the infrastructure to prove and build the pattern strategy; and then when the base mechanisms are tested, proving the reliability of what is done, they can be extended. Details can then be added inside the infrastructure following

its frame.



```
package patterns.examples;

public abstract class ParamDataGUI
{
  ProcessVariable p;

public ParamDataGUI(ProcessVariable p)
{
  this.p = p;
  if(this.p!=null)
  this.p.registerGuiPresentation(this)
  ;
}

public abstract void update(String s);
}
```

```
package patterns.examples;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;

public class GuiButtonUI extends ParamDataGUI
{

public GuiButtonUI(ProcessVariable p,
GuiDataButton data) {
super(p);

JButton btn = new JButton(data.value);
btn.addActionListener(new ActionListener() {

@Override
public void actionPerformed(ActionEvent e) {
   p.writeVar(data.value);
   }});
}

@Override
public void update(String s) {

  System.out.println("nothing to update
  (graphically)");
}

}
```

```
package patterns.examples;

import javax.swing.DefaultListModel;
import javax.swing.JList;
import javax.swing.event.ListSelectionEvent;
import
javax.swing.event.ListSelectionListener;


public class GuiListScroll extends
ParamDataGUI
{

JList<String> list = new JList<String>();

public GuiListScroll(ProcessVariable p,
GuiDataListScroll listdata)
{
super(p);
DefaultListModel< String> m= new
DefaultListModel<>();

for(int i=0; i < listdata.array.length; i++)
{
      m.addElement(listdata.array[i]);
}
list.setModel(m);

list.addListSelectionListener(new
ListSelectionListener()
{

@Override
public void valueChanged(ListSelectionEvent e)
{
  p.writeVar(Integer.toString(
  list.getSelectedIndex()));
  }});
}

@Override
public void update(String s)
{
 System.out.println("update list");
 list.setSelectedIndex(Integer.parseInt(s));
}

}
```

```
package patterns.examples;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JTextField;

public class GuiDefaultPanelBarUI extends
ParamDataGUI {

JTextField textField = new JTextField();

public GuiDefaultPanelBarUI(ProcessVariable
p, GuiDataDefaultPanelBar paneldata)
{
super(p);

textField.addActionListener(new
ActionListener()
{

@Override
public void actionPerformed(ActionEvent e)
{
    if(textField.getText()!=null && !
    textField.getText().equals(""));
    {
          /*user inputs contorol here*/
          p.writeVar(textField.getText());
    }}});
}
@Override
public void update(String s)
{
 System.out.println("update text");
 textField.setText(s);
}

}
```

Figure 3.49: Java code example of GUI components definitions derived from a common parent class.

The example in fig.3.50 shows implementation of parametrized factory pattern, through data structures objects that follow the same hierarchical logic of the components, using the described techniques.The same implementation can be used for process variable and converters.

```
package patterns.examples;

import java.util.HashMap;

public class FactoryGUIwithRegistration
{
static Object singleton = null;
HashMap<String, ProcessVariable> map;


public ParamDataGUI generateGuiElement(GuiData data)
{

ParamDataGUI anElement = null;

System.out.println("data code " + data.varCode);

       switch (data.Type)
       {
         case "GuiButtonUI":
         GuiDataButton btndata= (GuiDataButton) data;

         System.out.println("data code " + map.get(data.varCode));
         anElement  = new GuiButtonUI(map.get(data.varCode), btndata );

         break;

         case "GuiDefaultPanelBarUI":
         GuiDataDefaultPanelBar paneldata = (GuiDataDefaultPanelBar) data;

         anElement  = new GuiDefaultPanelBarUI(map.get(data.varCode), paneldata );


         break;

         case "GuiListScroll":
         GuiDataListScroll listdata = (GuiDataListScroll) data;
         anElement  = new GuiListScroll(map.get(data.varCode), listdata );

         break;

         default:
         break;
       }

return anElement;
}


public FactoryGUIwithRegistration(HashMap<String, ProcessVariable> map) throws Exception
{
  Exception e= new Exception("Singleton violation");

  if(singleton==null)
  {
        singleton = new Object();
  }
  else
  {
        throw e;
  }

  this.map = map;

  }
}
```

Figure 3.50: Java code example: parametrized factory method pattern implementation.

Obviously there is an order in the creation of these sets, due to their dependencies.

- The process variables have to be created at first.
- Converters need the access to all process variables during their creation and the mediator of this process, shall handle the data structures, with the information

of which components shall be bound together, during creation process, and properly manage the process variables set.

- GUI elements also rely on process variables objects and are independent from data converters. There is no need of being created before or after the converters, but converters can be used as standalone APIs to develop application. They should be created before as primary support for process variables.The generated presentation layer should be generated at last, leaving the option to develop an automatically generated presentation or use only the converters layer as an infrastructure independent from presentation layer on which to build applications.

These examples inherent to the construction mechanisms of the Java implementation are a way in which construction process can be explained. These examples obviously are far from the final result, which is very different in some parts. For example GUI components with more than one process variable needs a further extension of this mechanisms, developing a special case inside the factory, that has to bind the first variable with the depicted solution and then the derived class should have specific additional methods and fields to add the others. This could have been done in a more general way using a vector structure inside the construction process of GUI components (as done inside process variables to associate more GUI components), but cases in which a single component modifies more than one variable, are very rare and the implementation of a vector solution to reach more general methods could lack in performances due the fact that a lot of vectors will be created, most of the times to handle a single object. Inside constructions processes there are a lot of additional choices that can bring to an effective concretization of the same design strategy. It often happens that solutions are refined, improved and evolved. The important aspect is that this refinement is always coherent to the overall strategy or at least it should be. Not every aspect of design and construction is interesting for this kind of writing, because it should be focused on the innovative content, but it has to be reminded, that in the management of a process, every single aspect should be properly inspected even with a few drafts.

3.2.3.4 Protocol Mediator

The protocol mediator is simply the construction of the named pattern, applied to communication protocols. The mediator package contains the mediator object, often called "manager" or "handler". Regardless of the design strategies adopted for the protocol, the role is always the same, that is the creation of a bridge between protocol logics and process variables through the converters layer.

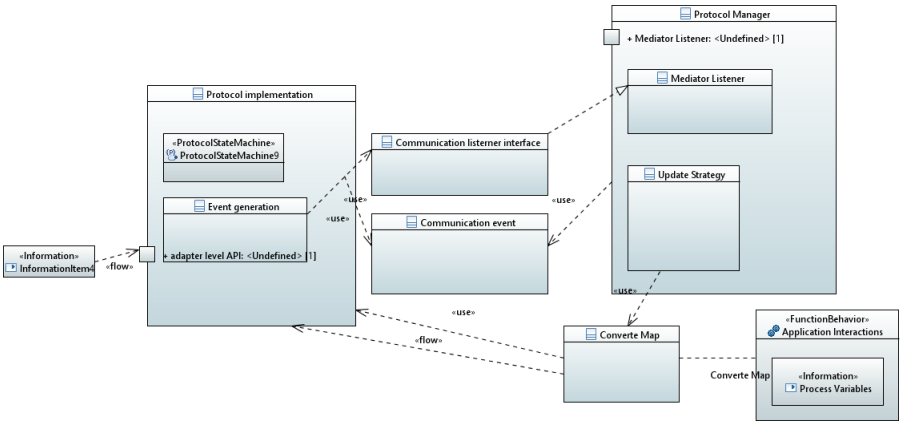


Figure 3.51: UML composite diagram of mediator implementation, designed with Papyrus tool.

Protocol data structures are based on the same common elements, such as structures like tables or arrays to organize data in which the variables are indexed through their standard that often are summarized in manuals to give a rapid access to them. The association between indexes, types and process variables is defined through protocol mechanisms which are the same for every quantity. The data converters contain the rules to translate these specific protocol access mechanisms. From the code examples shown before, it should be clear how these interact with the specific protocol; user input triggers the update of a variable, the variable triggers the converter which sends the data through the protocol object. The update operation has to be triggered by the mediator, using an update strategy. Usually messages contain information about indexes and this information can be used to retrieve the correct con-

verters or groups of them, when the update has to be performed. Even in this case, there are many options that can range from: update everything on every message or check indexes and update only the interested ones. These options can be constructed in many ways, using also additional structures that contain associations between indexes ranges and converters. The proposed example in fig. 3.51, is based on listeners construction which suits for event driven scenarios like the reception or dispatch of a message. Different protocols can be handled with different mediators in the same way,it is only necessary to create the proper converters and then the logic of the process variables is the same and is independent from the specific protocol.

3.2.3.5 Communication Protocol Libraries Design

The communication protocol libraries design and construction is one of the most complex operations that are faced in the engineering of these programs. The construction of the converters layer is guided by the defined class hierarchy, and has to incorporate the protocol functions. Design is different from one protocol to another, due to the differences in the adapters API, message formats and so on. Every library has a specific design which should consider the protocol aspects. The class diagram of the modbus protocol mediator implementation is given in the figure 3.52 below, which has been generated through object aid eclipse plugin, from the code of the actually working implementation. Communication protocols follow standards and their construction is ruled by them. It is not in the aim of this work to show how to code a working protocol library that should be a known task in the state of art. The important thing is to highlight the collocation of protocols inside the mediator pattern applied to this context, which is very useful to obtain the decoupling of variables from protocols, as claimed before. This achievement grants also a modular design in which libraries can be optimized and their design is self-contained in these mediator modules. Improvement of the libraries can be also developed as if they were a totally different and new protocol and then exchanged, after the development process.

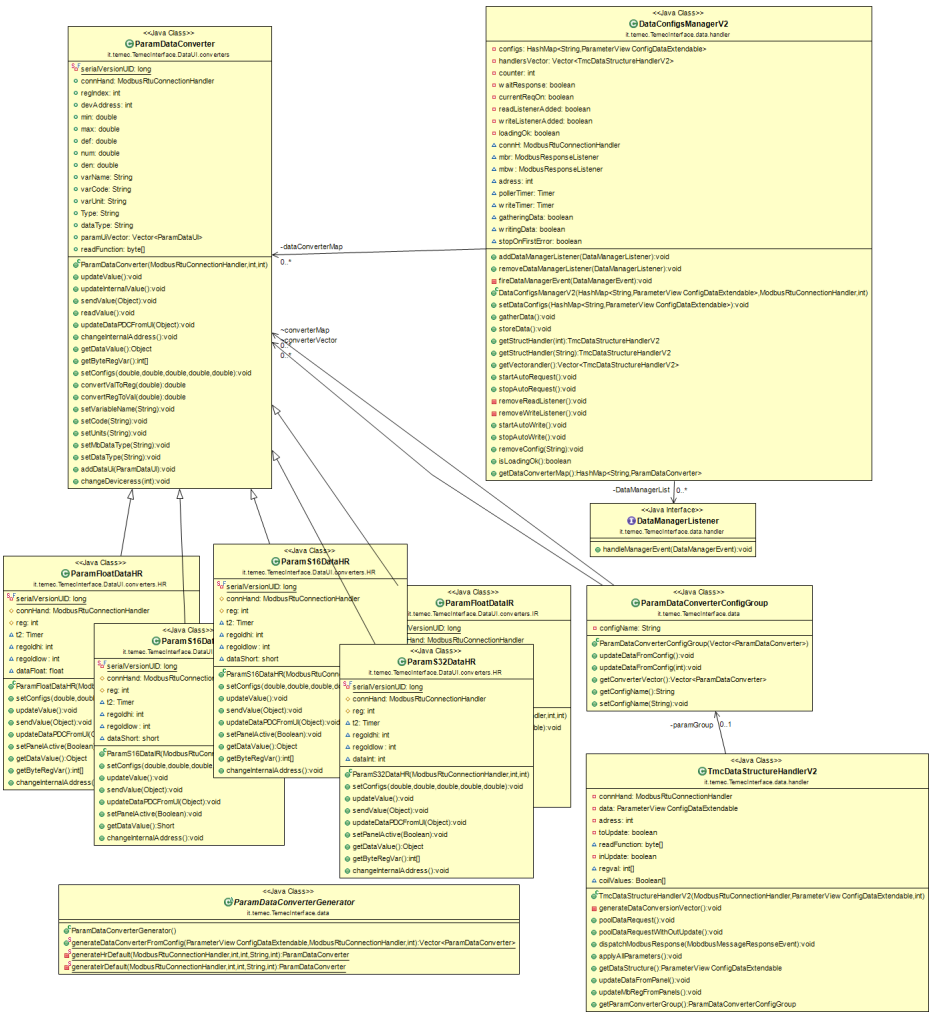


Figure 3.52: UML class diagram of Modbus protocol implementation, generated from the source code through ObjectAid Eclipse plug-in (Does not contain the representation of all the implemented classes).

3.2.3.6 Construction Improvements

There is only a nontrivial task that has not been contemplated in this design and construction, that is the possibility to operate with different protocols at the same time, synchronizing them on the same digital image. A possible construction could simply be the instantiation of a vector of data converters objects inside process variables instead of the assignment of a single one as fig. 3.43 and fig. 3.44. The registerPdc method should then implement an add operation on a vector. Write and update operations should then be performed on the whole vector. The part that has to be designed is the data coherence and propagation. The case in which two or more protocols are used together has never been faced and it is hard to think a possible use os such kind of solution, while the possibility of exporting the process variables image, obtained though a certain communication channel, and importing it, in another device and transferring it through a different protocol, is already possible, thanks to the process variables and converters layers.

More than one active protocol at once: the creation of different converter layers with the same factory concepts could be done as suggested few lines before, through the changing of the registerPdc method and the use of a vector. This implementation grants that messages that came from any of the active channels are sent to process variables and GUI elements in the same way as for the single protocol. Problems arise when the user tries to send data, which would be sent through different channels. The result will be coherent anyway and the device would write the same variable two times through different channels, operation that should not be dangerous but for sure is not needed. Anyway it is simple to develop a solution for that scenario, for example every converter object that belongs to a certain protocol could share a static enable variable through parent class. If enable is true the converters send data, if false they do not. Then it should be left to the user the possibility to activate a channel or not, through this enable mechanism. These solution have not been yet developed inside the framework, but this design can be applied in case of need. For now this scenario is not expected to be worth time and efforts.

Shared digital image: As stated before, process variable objects can be easily translated into the different protocols by changing the mediator. Process variables as described through the whole chapter, are stored in a dynamic data-structure and every variable is identified by a unique identifier. They can be easily exported to create configuration files from the parametrization of the device. As discussed in the extension of the features of the framework through the data structure in which the missing of a data does not compromise the system, but will only lack of the corresponding element, the same concepts is used for process variables. A digital image stored in a configuration file can be imported by every device that will only update and work with the variables which have the interested identifiers. If a device imports a file which has more variables than the ones present in it, some will be left to their default values. If there are more, some will not produce any update. If the unique identifiers are assigned properly and variables with the same meaning on different devices also use the same convention, it is possible to share the same configuration, even on different devices, because the digital image, is designed to be something that is abstracted from the specific implementation.

These last presented features can enable many application scenarios, especially for the shared digital image, which can be a shared digital configuration, that is something that can increase the interoperability of nodes inside the systems. It can also have common processes based on the same application logics, that are implemented on this substrate, which in my opinion is one of the most interesting result of this design. The construction dissertation should have clarified the design possibilities and the resulting benefits of the S.I.E.G.Fr.I.E.D and in general of the D.A.E.D.A.L approach.

3.2.3.7 Side Activities

As said in the beginning of the chapter, not all the construction aspects of the framework have been discussed, due to the fact that they are known art tasks, but to develop the complete system it is important to remember what has been done:

- GUI libraries.
- Data structures.
- Communication protocols.
- Graphical editor modules.
- File handling.
- Generator modules.

Finally, as said before when dealing with design, modules have to follow the pattern presented in section 3.2.2.2, but the design of the features that these modules implements, depends on the feature itself and the same is true for their construction.

3.3 Software design: comparison between "traditional" and D.A.E.D.A.L approaches

The development of these kind of programs, even with a structured approach, fig. 3.53, imply the handling of protocol mechanisms like write and read requests, and the data indexing handling. Even with the use of high level functions, handling a single variable, needs the implementation of an update mechanism in which the proper function is called, the reconstruction of the variable from transmitted bytes and the composition of the message to send its value. It is hard to develop solutions with less than four or five function calls with the corresponding parameters. In this case is considered only the simple read and write case, without labels, scaling factors, range checks and all the properties that have been proposed for process variables. Adding these properties needs at least their declaration in program variables, assignment and manipulation. Only considering a basilar functioning, in order to handle a variable, there is the need of inserting at least three parameters: device bus address, map index and a value. Even considering a single function call addition when a property is added, the needed code grows linearly with them. To build a simple graphical component for a user interface a significant number of lines of code are needed as shown in

the example in fig. 3.54, which is also a very simple component. Even reusing these classes is hard to limit the numbers of lines of code that are needed to handle them. Assuming, in a very optimistic way, that to handle a variable, more or less ten lines of code are needed with the same number of parameters and that the protocol management allows to handle these insertions without any problem, it is easy to understand that even the handling of few variables requires a significant coding activity.

```
//translate the variable into a protocol message and transmit it

public void sendUint32Mb(long longc, int devAddr, int varIndex )
{
   int regs[] = new int[2];
   regs[0] = (int) (longc & 0x0000FFFF);
   regs[1] = (int) ((longc & 0xFFFF0000) >> 16);

   getModbusConn().sendModsbusMessage(WriteMultipleRegister(devAddr,
   varIndex, regs));
}


//convert protocol raw data to a variable

public long get32VarMb(int devAddr, int varIndex)
{
   int hi  = getModbusData().getDeviceData(devAddr).
                   getHoldingRegisterVector().get(varIndex+1);

   int low = getModbusData().getDeviceData(devAddr).
             getHoldingRegisterVector().get(varIndex);

   long rawLong = (long) (((long)low)|( ((long)hi<<16) & 0xFFFF0000));

   return rawLong;

}

// read data from device

public void readMbData(int devAddr, int varIndex, int quantity )
{
   getModbusConn().sendModsbusMessage(readMultipleRegister(devAddr,
   varIndex, quantity));
}
```

Figure 3.53: Example of variables handling with high level protocol functions

The D.A.E.D.A.L approach, both through the use of the automatic interface generation and data converters layer is able to reduce or totally avoid, the insertion of new lines of code. The variables properties definition and mapping require the insertion of a single word or value for every field inside the editor. The creation of graphical components requires only a few mouse clicks inside the editor. The reduction of the time

```
public class ExamplePanel extends JPanel
{

 . . .

  public ExamplePanel()
  {
  setBounds(0, 0, 250, 100);
  setLayout(null);

  JButton btnSendBtn = new JButton("SEND");
  btnSendBtn.addActionListener(new ActionListener()
  {
     public void actionPerformed(ActionEvent arg0)
     {

        long longN = (Long) Long.parseLong(textFieldValue.getText());

        sendUint32Mb(longN,1, 54);
     }
  });

  btnSendBtn.setBounds(150, 30, 85, 35);
  add(btnSendBtn);

  textFieldValue = new JTextField();
  textFieldValue.setBounds(10, 35, 115, 20);
  add(textFieldValue);
  textFieldValue.setColumns(10);

  }

}


//On some event update variable

{
  textFieldValue.setText( get32VarMb(1, 54) )
}


//Trigger an update requestr when needed or through a timer

ReadMbData(1, 54, 2 );


//process adn receive messages

public class MbConverterFunctionalGroupHandler implements ModbusResponseListener
{

   ...


   public void dispatchModbusResponse(MobdbusMessageResponseEvent modbusMessage)
   {


            if(modbusMessage.getCrcOk())
            {
                  modbusMessage ...
            }

   }
}
```

Figure 3.54: Example of a simple graphical component and variable handling

needed by these operations is not only evaluable by the kind of operations involved and their number, this approach grants a guided design and more reliability, that also mean less time spent in maintenance and "bug" fixes. In my opinion the benefits are naturally clear for a person who has a minimal knowledge in programming and related issues and also with these examples proposed here and in this writing, it should be easy to understand the implications of this work.

Chapter 4

Firmware Design for Embedded Industrial Devices

In the introduction chapter, many aspects of what are the elements inherent of embedded device programming have been already discussed, technologies, in fields of applications, devices... In this chapter a more detailed overview is presented to introduce the firmware components of this framework. Devices complexity can be different, as said before, there are devices that can be composed only of analog component and in a minimal part of digital components, also without computational capabilities, and on the other side there are devices with huge processing capabilities and a consistent part of them is constituted by program logic. In this case the attention if still focused on these devices with at least a central processing unit (CPU) and communication capabilities that need a proper commissioning software for their usage, such as the case of electric drives, for motor and machines control.

4.1 State of the Art of Firmware Engineering

Firmware engineering follows the same process as the software engineering, described in section 3.1, with the difference that firmware programs do not use object oriented languages, so the practices that have been presented for the software, usu-

ally do not fit this context. Embedded devices, most of the times implement real time applications, in which computational time has to be strictly deterministic. Even in the case of the presence of an RTOS, object oriented languages are not used so often, due to the fact that they rely on mechanisms that do not provide real time capabilities. They can be used inside real time operating systems for object oriented languages that have proper support for them, but they need to rely on the OS support to achieve real time capabilities and need elaborators with enough computational power and memory to support it. Embedded devices are usually programmed without operative systems or when they are, they use embedded RTOS which are slightly different from the computer ones. This work wants also to underline that, due to the increase of devices capabilities, the firmware complexity is arising in its evolution. To face this scenario proper design and construction techniques can be developed, also through heuristic approaches as the ones that have been matured in the object oriented programming. Firmware and software elements are often treated in the same way, but when dealing with design and construction aspects, they need to use different methodologies, proper for the different instruments, that are used in the development of these two categories of programs. This chapter presents some developed techniques for the most common development scenario that has been encountered in my experience, that is the development of firmware programs, written mostly in C language, for motor control applications. In past years firmware complexity was more inherent to algorithms implementations on micro-controllers [71], nowadays, due to the growth of these programs, their structure and development is adding more elements in the overall complexity. Programming faces many aspects of devices behaviors, from boot-loaders and program updates[72], to recently added ones, like security [73].

"The structured testing of software (and thus firmware) has grown to be an important discipline on its own right. Primarily the complexity of firmware requires rigorous testing to minimize the potential business risks of embedded software systems [74]."

The "C" language is one of the most used programming languages in the embedded field. The ranking of IEEE spectrum [75], reported in fig. 4.1, put the C language at third position of the embedded category. First and second positions are

held by object oriented languages, which can not be used in the programming of micro-controllers except rare cases. Usually object oriented languages, that are applied in the embedded fields, are used for the realization of software programs. So the C language can be considered one of the most, or even the most, used language in micro-controller firmwares development [76].

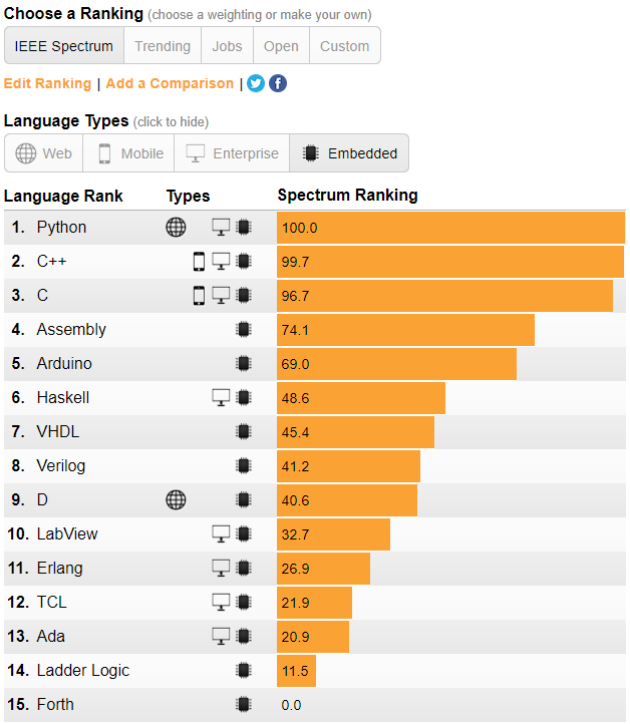


Figure 4.1: IEEE Spectrum ranking of the most used programming languages.

C language seems also to be widely adopted by industries, in the programming of micro-controllers and DSPs, due to the many development environments that manufacturers distribute to program their chips, like the eclipse based ones, that have been introduced in section 3.2.3.1, and many others, like Keil or Atmel Studio, based on different C-compilers, like various GNU toolchains, Microsoft ones or other proprietary implementations, plus a lot of SDKs, libraries and protocol stacks, that support

application development, in this language.

The C programming language is a lower level language respect to object oriented ones, but has some structure mechanisms and the possibility to create file hierarchies and libraries. It is also used to integrate specific instruction sets inside its files. It can work with memory references and values. These characteristics are probably among the reasons of its diffusion in embedded programming, due to the low level of abstraction, that can be used to produce essential routines, with low overheads, but also offers the possibility of structuring program code, maintaining a good trade-off between features and performances. As happens in software products, when complexity increases, handling and developing products, can become a time and efforts consuming activity, and the need for strategies to face this situation starts to be felt [77] Devices can have different features, that can be used to classify devices into

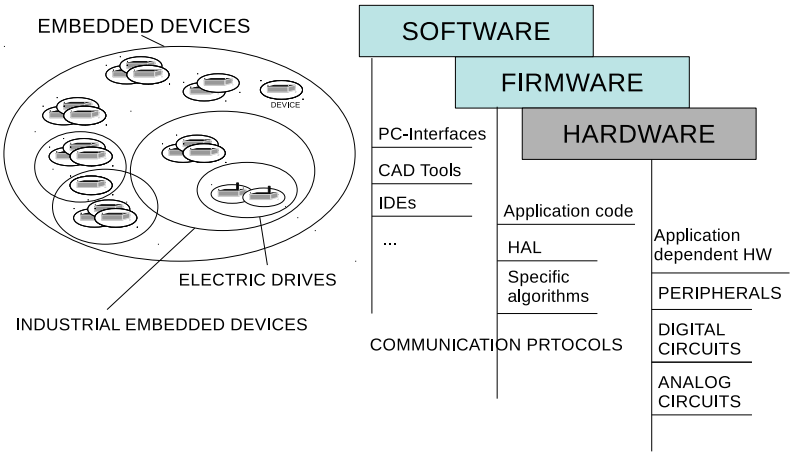


Figure 4.2: Embedded devices categories and composing elements.

categories, see fig. 4.2.These categories can share libraries to implement their functionalities. There are already many studies on how to increase reuse of code and other design aspects, in many categories of embedded devices, like designs for Internet of Things (IoT) devices [78]. Examples often refer to "universal firmware for ..." [78], [79], and then refer to the specific category of devices that they should deal with.

SDKs are also examples of "universal firmwares", meaning that there is a common ground for some categories that can be the basis for many applications. For examples the MCUxpresso IDE developed by NXP has also a tool named SDKs builder, which is an instrument that assembles a correct package of libraries for a specific controller, or even evaluation boards, with all the functions to correctly handle peripherals and circuits.

Also for example a user can compose the SDKs of two different boards with two different micro-controllers and for both of them make a "blink-led" program. In both cases inside the IDE the user will have access to a function named like "toggle_led" plus something that identifies the specific led, totally masking what are the hardware circuits and the instruction set involved, that make possible to change the state of the I/O module.

These SDKs often have libraries that use the same function definitions for similar controllers in order to totally mask the hardware specific dependencies. Changing then device and libraries, the function call inside the program, will be exactly the same even in its syntax, such as the case of the Kinetis SDKs, which have different cores and peripherals, but these are deputed to similar applications, and their SDKs, are built in the described manner.

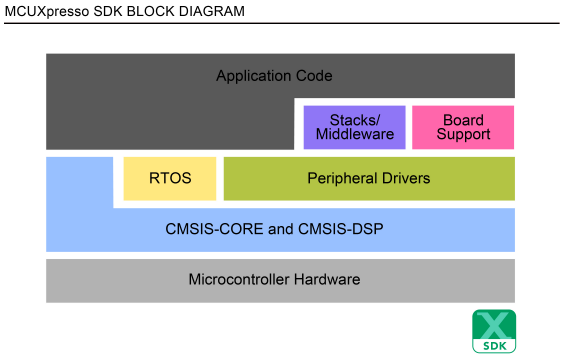


Figure 4.3: MCUXpresso SDK block diagram.

For example, DSPs based on ARM Cortex M4 ans M3 cores, equipped with different PWM peripherals can configure the same behaviour for a specific channel us-

ing the same function call, but in case of significant differences between the two hardwares, these functions often need also some configuration parameters. So the resulting code could be somehow different, but the reuse and interoperability capabilities are a great achievement for these manufacturers. The example in fig.4.3 is taken from the NXP products, due to my personal experience and usage of their controllers in many projects. There are also many other companies that develop such solutions.

This abstraction process, that often tries to mask the hardware complexity, can also exploit its functionalities in an efficient and comprehensible way. The result of this process, is often called Hardware Abstraction Layer (HAL). There are many ways in which it can be realized, but this concept is quite always present in programming contexts, as can be evinced by examples in fig. 4.4, fig. 4.3 and also from literature [2]. Many others examples of abstraction layers can be provided, for OS, firmware development and so on, but to avoid excessive details in this subject, that is not the core of this writing, further researches about that, are left to the reader.

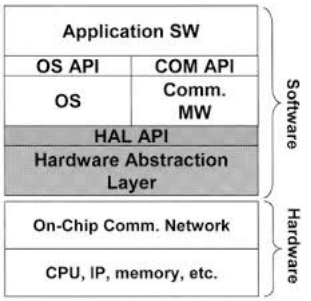


Figure 4.4: Example of Hardware Abstraction Layer (HAL) [2].

In section 3.2.3, about the software construction, it has been already highlighted the importance of establishing proper methods in the design and construction of software programs, that is also true for firmware ones. In this case, the problem is even much more felt; in the case of software, especially OOP-based, the design practices thet have been described in the previous chapter have been developed. The UML has been proven to be a valid support for modeling and design patterns are deeply

studied in literature. In the firmware environment there are attempts to introduce the same methodologies, often trying to adapt software processes to the firmware ones [80],[81], especially in the modeling. UML, which is the most used and widespread standard, is based on object oriented paradigms, that generally are not suitable for embedded devices, except when proper RTOS can be used. Strategies and models should be applied according to the paradigms they follow. For example, using class diagrams to model a program written in “C” language, that does not have classes, could not be a wise choice, while a state diagram could be clear and appropriate to model a finite state machine that executes a task inside the program. These considerations do not want to underestimate the work of some books, like DESIGN PATTERNS FOR EMBEDDED SYSTEM IN C [80], which face many problems of the embedded systems, but I think that proposed approaches are better suited for RTOS or similar platforms, due the object oriented design. The author shows how to build an object oriented structure with C language, that is something that I can not say that is not interesting for certain solutions, in which this kind of construction can be used, in order to apply object oriented patterns, such as the known software ones.

"While C is not an object-oriented language, it can be (and has been) used to develop object-based and object- oriented embedded systems."[80]

Concerning my personal experience, especially in industrial environment, the use of embedded RTOSs is not often a popular choice, due the optimization processes that cut resources at the minimum in order to reduce costs. The Bare-Metal/Bare-Machine/BareBack approach have been more often encountered. All these terms refers to a core executing instructions directly on logic hardware without an intervening operating system. In facing these scenarios, there is the need for different techniques which could be also used inside RTOSs which should be a further improvement of the available possibilities. In many examples of these works [80], [81], there is also the presence of hardware delay instructions, or "busy wait" synchronizations, which are sometimes used also inside some HALs. These mechanisms are sometimes necessary to realize the proposed logics and some of the object-oriented

ones, but they affect performances. Often the realization of this object oriented structure needs more hardware resources than would require a development without it. Especially for motor control applications, in which most of the control algorithms have to be executed in less than a hundred of microseconds, this can be a huge issue. So the aim of this work is to introduce some techniques that can be used also without an object oriented or similar infrastructures, trying to maintain a design which can achieve abstraction from hardware and maximize reuse of code, even in constrained environments. These methods can be also used where these constraints can be relaxed, as the case in which an embedded RTOS can be supported. In the case of real-time controls with such constrained execution time, the dynamic handling of memory, the "OS kind", simplifying the concept, is something that has often to be sacrificed. Several studies, surveys and reports from experts in the field claim the importance of a structured approach to design and modeling, supporting evidence that there are some aspects that have to be further analyzed [82], [83]. The general aspects of handling a program are well defined, but not enough to cover the whole process. Firmware and software are often referred to as the same thing, that can be reasonable to a certain level of abstraction, but when approaching design and construction aspects, constraints, instruments and methods, even for devices with operating systems, are totally different [80]. For these reasons, the two definitions cannot be collapsed into only one.

This chapter presents some methods inherent to firmware design and construction that are intended to maximize reuse of code and contribute to a structured approach to these stages of the firmware development. These methods are part of the D.A.E.D.A.L approach, of which are its firmware part. These mechanisms are realized using C language, which is the one that has been used to develop the firmware libraries of this framework. C languages has been choosen due to its diffusion and the many available SDKs and third-party libraries used in firmware development.

4.1.1 Embedded Device Programming

In the introduction some of technologies that are part of elaboration units were presented. Industrial embedded devices, regardless of the technology used, FPGA, DSPs

or both, have at least a processing unit, program and data memories and hardware peripherals to handle. In the case of more processing units, proper strategies and patterns shall be considered, but for now, the attention will be focused on methodologies for the single unit, which could still be a valid base for the multiple units scenario. The presence of elaboration units needs programs that implement the application logic.

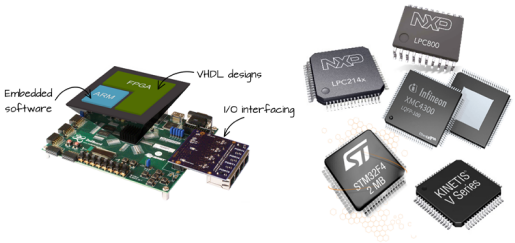


Figure 4.5: Zedboard: FPGA plus ARM core on the left, ARM core based DSPs on the right

In the above figure 4.5, there are different examples of the technologies that execute firmware codes; Zedboard from Diligent company is an example of reconfigurable hardware that relies on a core unit for the processing capabilities. There are also some controllers from NXP, Infineon and ST, many of them having an ARM core, and some others with different core architectures. The relevant consideration is that most of these devices use programming environments based on C language for their firmwares. As should be known, core architecture is what defines the instructions set of the elaborator, and also its assembler instructions. Programming logics using assembly code is no more a common practice except in rare cases. Firmwares are written mostly in C code and in some parts, of instructions that are architecture specific. These instruction are kept isolated as much as possible, in order to avoid excessive dependencies inside the C portable code. Programs are composed of pieces of code, more or less dependent from the hardware platform. The part which is not dependent, is the pure C code that can be interpreted by compilers 4.6. In this case, the example shows C code that is translated into programs instructions for the various controllers, through compilers. The hardware dependencies of code is what the

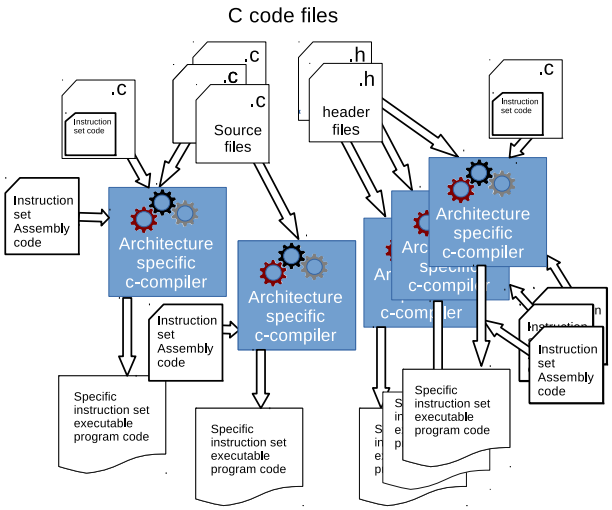


Figure 4.6: C program compilation for embedded devices.

HALs try to mask and exploit. The application logic part is usually composed using portable code at most, with few or no dependencies, as suggested also in figure 4.3.

4.1.2 Algorithms Implementations

Everything that is related to the elaboration of a computer or a micro-controller is based on binary logic (and will be in that way until new technologies, like quantum computers, will extend the physical symbols range, but nowadays a common architecture supports only zeros and ones). This is another element of the complexity of embedded programming. In the development of the programs of these controllers,it often happens that mathematical models have te be translated into this logic.

Discretization : it is the process of transferring continuous functions, models, variables, and equations into discrete counterparts. This process is usually carried out as a first step toward making them suitable for numerical evaluation and implementa-

tion on digital computers. This process involves many other mathematical concepts that are inherent to discretization, like quantization and sampling. All these concepts are part of the fundamentals knowledge areas of computer science, which are briefly introduced to give the overview of the present complexity aspects of firmware development and also to introduce some specific work that has been done inside the framework. Algorithms implementation is something that has been deeply studied in past years and this activity is also supported by effective modeling diagram, like state machines and flowcharts.

4.1.3 Industrial Embedded Firmware Design

At the base of reuse of code there are strategies and a proper construction for reuse. In order to define patterns, it is necessary to understand common elements and their aspects. These aspects have to be handled in a proper way, in order to achieve desired behaviors.

The general behaviour of this kind of devices has some common characteristics:

- When powered, "boot" the program, from memory or communication channels;
- Initialize the memory state to a default condition;
- Load parameters and state from memory which can be the internal memory or an external source, depending on the memory storage capabilities of the device;
- Configure the peripherals;
- After these stages, start routines to executes application tasks.

This behaviour is intrinsic in the realization technology of every controller. Two of the most important modules inside the core of the device are the interrupt controller and the system clock manager; through their configuration, these units activate a set of periodic or event-triggered subroutines in addition to the main execution flow. It is common to have periodic routines together with asynchronous ones, both handled with a priority mechanism and enhanced memory access. In addition to that, devices are integrating several communication features, with one or more protocols on different physical media. Structured design and construction shall consider the structural and behavioural aspects and how to represent them for modelling and documentation. This domain describes a wide class of devices in their context. It is common to have periodic routines together with asynchronous ones, both handled with a priority mechanism, enhanced memory access and caches. In addition to that, devices are integrating several communication features, with one or more protocols on different physical media.

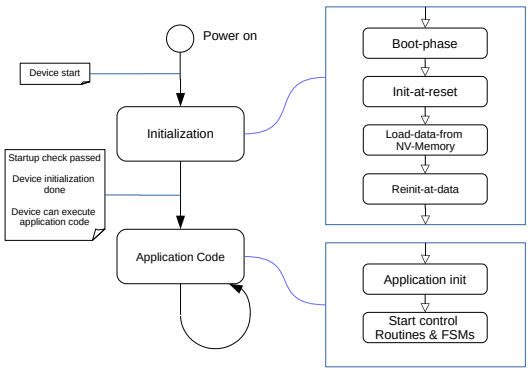


Figure 4.7: Generic state machine of an embedded device processing unit.

A generic description of the tasks that devices have to do, is something that can be represented through a flowchart, see fig. 4.7, in which the last stages are inherent to the execution of the application code, which is the purpose of these devices. Previous stages are the necessary actions to reach this achievement. All the phases in this flowchart needs to be handled by a dedicated Finite State Machine (FSM), that calls routine and initialization functions, when they are needed. This FSM also handles the evolution of all the other FSMs. This description is used as base for some design strategies which can be applied in the overall structure of any firmware program, in which stages can be complex finite state machines or simple instructions or just a jump to next state, depending on device capabilities. Industrial devices are often used to control physical quantities to operate manufacturing processes, which classical automation scheme is the one has been introduced in section 1.1.0.1, which is represented in fig. 4.8.

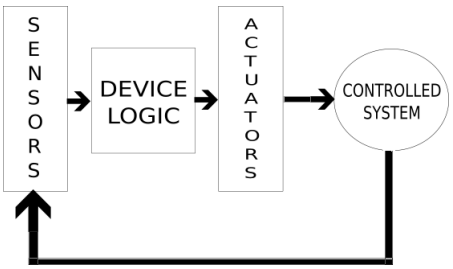


Figure 4.8: Closed-loop control.

4.1.4 Firmware Design for Electric Drives

Electric drives, such as the AZ3, fig.4.9, made by TeMecDrive, are a particular category of industrial embedded devices, often used in complex machinery, such as axis motion machines, CNC machines, robots, lifters and also simple motions, like automatic gates. Modern drives, have many features. They use one or more industrial communication protocols, handle different input sources and offer many functionalities. They are almost always combined with a proper commissioning and monitoring software. AZ3 in one of drives in which some of the presented techniques have been applied to.

Figure 4.9: AZ3, low voltage drive for AC and DC brushless motors.

There are different motor topologies:

- DC Motors
- Permanent Magnets (P.M) Synchronous Motor
- Induction Motors
- Stepper Motors
- Reluctance Motor
- Many others...

Which differ from one another by the number of stator phases, rotor composition, presence of magnetic elements on the rotor or on the stator and mechanical or electrical phase commutation mechanism. The discussion about electric motors is not in the aim of this work and is also a very vast subject. The important concept is that they are machines that convert electrical power into mechanical power and vice-versa. Electric motors are driven almost always by a proper proper power electronic drive, that is usually able to handle only some types of motors for which it is designed and is able to interface with some sensors that gather information about the motor state, that combined with a machine, is the system to control.

4.1.4.1 Motor Control

Firmware for motor control depends on the type of the motor itself, which can have different physical properties that are used to generate motion, but due to the fact that it is always a matter of electric power, every motor can be controlled by driving voltages and currents on its access terminals, that are made for that purpose. For example, in fig. 4.10, there are two three-phase brushless motors, produced by TEM Electric Motors, and a representation of the internal section that highlights the stator phases and four pole-pair geometry of the permanent magnet composed rotor. If vector control is used, it is often based on Clarke and Park mathematical transformations that permit to work with an aggregate model in which a reference current is directly proportional to the motor torque as happens for DC motors, in which there is only one wiring. Through these mathematical instruments and under some conditions the mechanisms

that control the torque current of a brushless motor, are the same for a DC one. Many topologies of motors are controlled through two currents and two voltage references which are torque and flux components of the electromechanical forces that generate the motion. Then, how these quantities are driven, depends on the motor model.

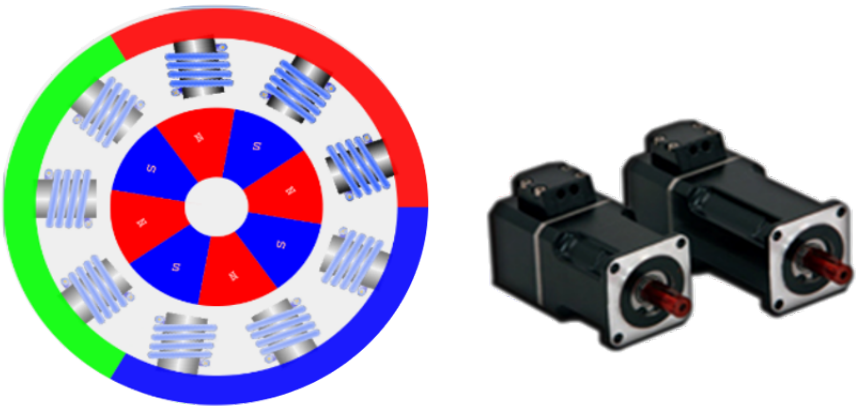


Figure 4.10: On the left, a software GUI component representing three-phase brushless motor with four pole-pairs. Brushless motors produced by TEM Electric Motors on the right.

Through sensors, or sensorless techniques, it is also possible to monitor and then control other physical quantities, like speed and torque, that directly become part of the controlled system connected to the motor. Also in this case there are common elements that can be evinced by these descriptions and from the ones in previous sections.

The knownn state of the art about motor control is divided into two main categories which are:

- Cascade Controls, which is an evolution of traditional control based on the P.I.D controllers.

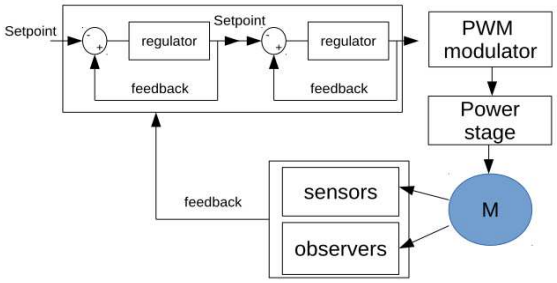


Figure 4.11: Closed-loop cascade control model.

- Model Based Predictive Control (MBPC), which is an advanced method of system control, that relies on dynamic models of the system, most often linear empirical models, obtained by system identification.

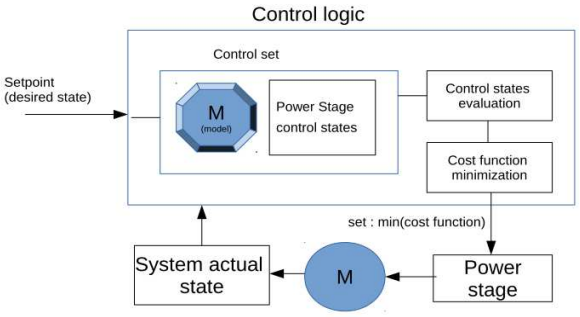


Figure 4.12: Closed-loop MBPC model.

The cascade controls are the one that I have used in the development of firmware

for motor controls applications, which are older methods than the MBPC. Even if they are a deeply explored subjects there also opportunities for improvement. MBPC seems a promising method, but is still hard to encounter on field application in which companies are interested (at least for my working experience), maybe because MBPCs are often computationally intensive and also because the model of the controlled system has to be known with great precision for them to work properly. So the efforts in implementing practical solutions have been focused in traditional cascade controls.

4.2 Firmware Architecture

This chapter presents the firmware part of the D.A.E.D.A.L architecture, which is more a matter of design guidelines rather than an effective structure. These guidelines, impact on the construction of firmware to reach some advantages that will be clearer in reading the following sections.

It is essential, in the design of programs, to have reference points to orient inside its lines of code. Without these references, when programs grow in number of files and lines of code, it is very difficult to insert new features and it is easy lines of code that can origin unwanted behaviors. Even for programmers with a very good memory, it is hard to keep trace of the whole structure (also very risky) and in the case of teams of developers, it is very hard to coordinate the development without precise references, especially for software, which is expected to grow a lot more than firmware.

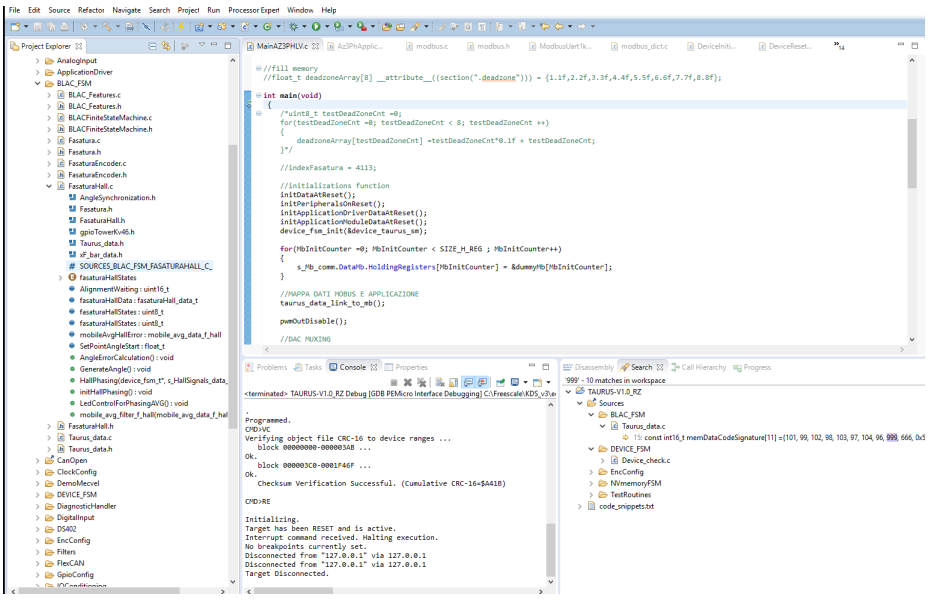


Figure 4.13: Firmware project view, folder and files, inside Kinetis design studio IDE

As can be seen in fig. 4.13, projects can contain a huge number of folders, that

can incorporate several files, and every file can range from few lines of code till hundreds of them. Especially for people that did not wrote these lines, it is not easy to understand the whole behavior, that is the reason behind software quality studies and standards, the importance of design documentation and rigorous testing. Every line of code has to produce the correct program sequence and evolution. The architecture design is the map that allows to understand the program structure, in order to met the essential concept highlighted few lines before. The proposed techniques have been thought also to help the characterization of these conceptual maps.

4.2.1 D.A.E.D.A.L Approach

Most of the D.A.E.D.A.L aspects have been already shown in previous chapters, especially for what concerns the data centered design strategy. Returning to this concept, with the aid of fig.4.14, to remind that the S.I.E.G.Fr.I.E.D software framework, generates through the editor program the application data file and the source files for protocols data maps that are added inside firmware program. Framework elements interact through data, so if the design of firmware part is based on data interfaces they can be easily integrated inside this approach. Due to the fact that all the devel-

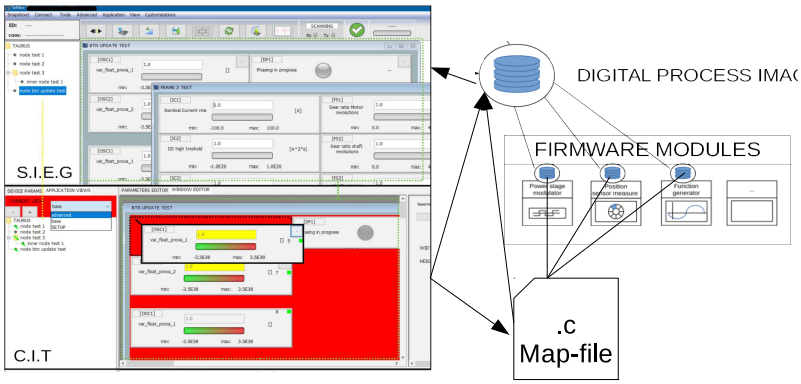


Figure 4.14: Data structure centered design, firmware perspective inside the framework.

opment is handled by the framework it is possible to define the way in which code maps are generated and also with a proper firmware structure it is possible to integrate them with minimum efforts. The following sections present how to construct these modules and interfaces, to create a modular firmware structure based on C language mechanisms, intended to maximize reuse of code and tested design patterns, without the use of object oriented methods. Its application to electric drives is a valid proof of the capabilities of these techniques, and this has also provided a real application scenario for them.

4.2.2 Firmware Design

The development of firmware that executes directly on cores, often called "bare metal" programs, is a scenario that could be improved by proper methods and modeling techniques. As stated in the description of the state of the art, when object-oriented techniques cannot be applied, it is often hard to find the proper instruments to approach design and construction in a structured manner. The aim of this work, is to propose strategies for this scenario, through the proposal of patterns, methods, models and examples which have been also applied in the development of products. As already introduced the algorithms implementation is something that is still a non-trivial task due to many mathematical aspects involved in the digitization process, but there are formal methods to translate models into algorithms Flowcharts and finite state machines (FSMs) diagrams, are valid tools to model their structure and composition, also for C language. It is common practice to represent execution flows and algorithms thorough the use of these diagrams [84],[85], [86]. What is probably missing now, in the firmware development, is how combine all the features inside devices and how to build an infrastructure for them.

4.2.2.1 Device State Machines

In section 4.1.3, referring to fig. 4.7 is presented a generalization of the stages that an embedded device usually encounters before executing application code. There are two main phases of the device, the first one is inherent to the startup in which the

device is powered up and its internal circuits are activated in a proper sequence. The second one it is when is able to execute application code. The proposed flow, is what normally happens in a firmware program, and as said before, these stages can be different from a device to another, due to their capabilities. After powering up, they have to load the program code, which can be inside the internal memory or transfered through other sources at the start-up. In this stage the upgrade of the firmware can be also done.

Boot stage : In the boot-loading stage, a proper program, that resides in different program memory area than application code, is deputed to the transfer of the firmware program bytes in the program memory area of the controller.

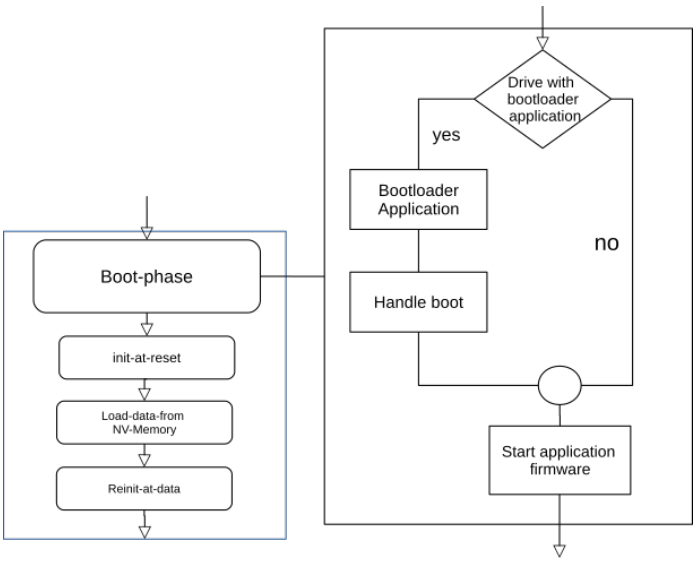


Figure 4.15: Boot phase flowchart.

AS shown in fig. 4.15, this process could be a simple jump to the first instruction of program memory area or the execution of a complex program which handles the update and loading of application code. In these examples it is important to show how nesting different FSMs inside a more general one can create a common structure in

which device capabilities are blocks inside states of FSMs. These stages are handled through separate FSMs in which every device executes proper tasks according to its capabilities. FSMs are very useful in modular approaches because they can create flows of execution with branches and reunions in which every state executes some actions. The boot phase ends in the same way for every device; the firmware program is ready to be executed.

Reset initialization : After the boot stage, the program code changes the memory state with the default values that are written in the program itself, defining its initial state. This is a very important thing because, programs are FSMs with memory, their future state depends on the current one, so the memory configuration influences the execution flow. Some environments provide automatic routines in which data that is not initialized (that are memory areas expected to be accessed by the program), is initialized to a default value. The default value is necessary to have a deterministic behaviour after the reset or power-up of the device.

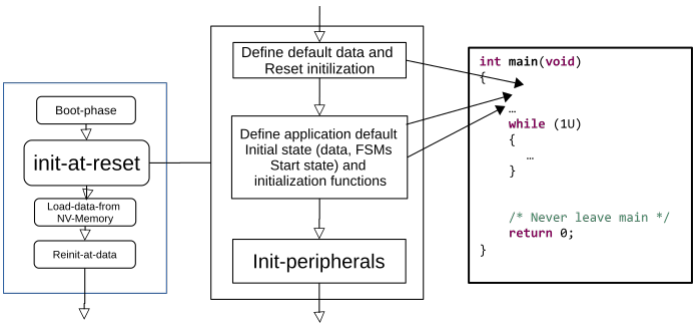


Figure 4.16: Reset initialization flowchart: tasks and program analogy.

This stage usually comprehends a series of instruction in the main execution flow of the program before entering the endless "while-loop", cycles the contained instructions waiting for interrupts, until device is shut down or a reset condition occurs.

Loading data from memory : After the initialization phase, which is done before entering the cyclic operation condition, it could be necessary to retrieve device

parametrization from a non volatile source of memory, that should be used as initial state for the application code. This stage often needs the configuration of some peripherals. For example, to be able to interface with external memories, reason why it is done after an initialization stage in which needed peripherals can be configured.

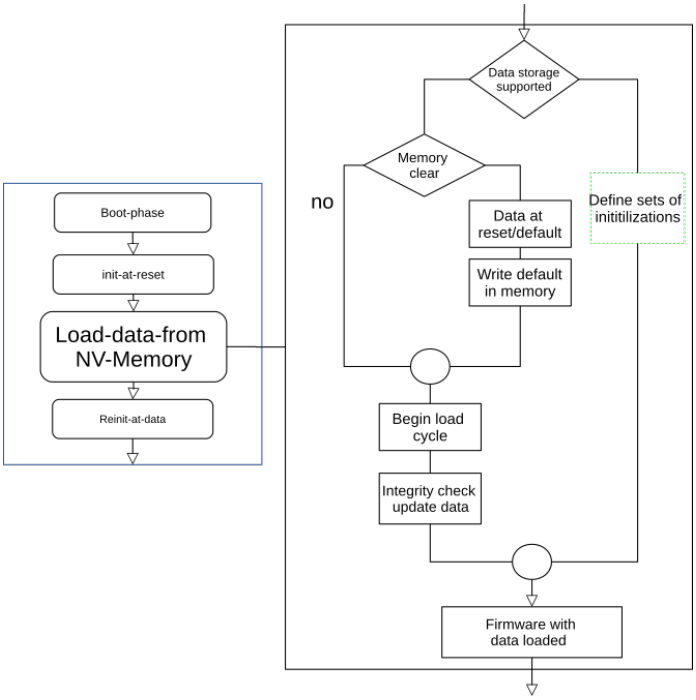


Figure 4.17: Load data from memory flowchart.

Fig. 4.17 shows two commons scenarios, devices with permanent data storage capabilities and devices that are not able to memorize configurations. In the second case, this stage simply defines the default application state, as the reset operation or a different settings.

Reinit at data : See fig. 4.18 This stage is called reinit, due to the fact that after loading configuration of the state of the programs, devices could need to reconfigure their internal hardware in order to activate specific behaviors.

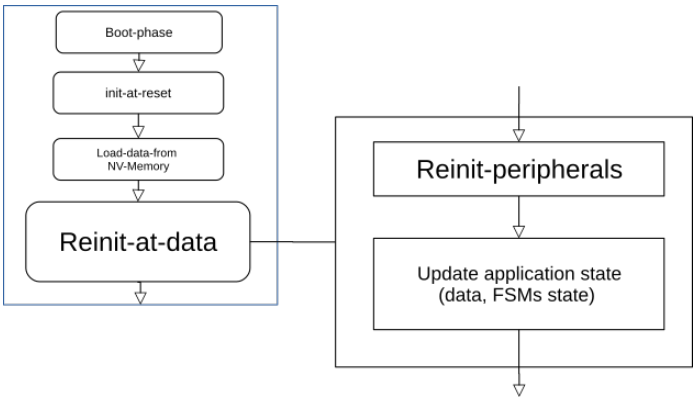


Figure 4.18: Reinitialization flowchart.

Application Code Execution : After these stages, the main execution flow of the device usually enters the "while-loop" and cycles the contained instructions. Once done that, usually all the necessary interrupts routines are activated and periodic routines start their execution. Devices deputed to the control of systems often have a

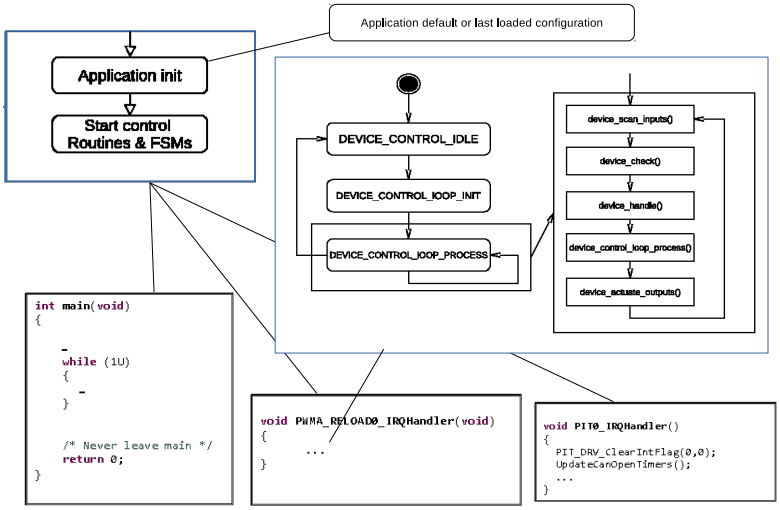


Figure 4.19: Application code: flowchart, control loop FSM and routines.

periodic task with the highest priority which contains the control-loop, and other routines that are executed with lower priority or with asynchronous mechanisms. Fig. 4.19 represents the last two stages of the proposed flow, which are inherent to execution of application code, the first one simply initializes application code to the default application state. Then the main execution flow starts its cyclic routine and the other subroutines are activated. In the figure is also proposed an additional FSM contained in the high priority routine which is deputed to the control of the system. The definition of two separate state machines, one for the main execution flow and one for the control loop, grants the possibility to decouple their evolution. The main state machine is also able to interact with the control one to activate and deactivate the control features using the "idle state" of the FSM. Every execution flow has a

FSM which follows its evolution that is independent from the others, but they can also interact. Their implementations and also initialization stages, can be done inside separate files and modules, in order to realize a program structure that follows the device behaviour. This is very useful, especially for testing, because FSMs can give informations about the operations that are executing, in case of faults or other interesting events. Except this benefit and a possible common and modular structure for different devices, there are no particular advantages in adopting this design, but it is something that could help also the understanding of the overall structure and behaviour of the device. The application of the same structure to different devices needs an appropriate construction which can be done in several ways, but due to the high dependencies of the hardware platform, in particular for initialization code, it is better to use this design only as a guideline, rather than as implementation of actual FSMs with different methods for different devices. For example, always considering C language, a generic FSM could be implemented declaring prototypes of functions inside states, and then defining different functions in different source files, one for every device, in order to compile the proper one, through conditional compilation. Using conditional compilation to implement prototypes with different codes is a common practice for C programmers. The only drawback is that it can be done only at compile time, differently from object oriented logic in which this behaviour is done even at runtime through the polymorphic feature.

4.2.2.2 Abstraction Layers

The abstraction is something that has been already depicted inside this work, HALs are the program parts that are intended to mask hardware complexity, that build a substrate for application code that should be independent from it. The effective decoupling is always a matter of construction, in special manner of the construction for reuse [87]. There are many ways in which HALs can be constructed. In the case of C language which is based on the structured and procedural programming, it can be done, as said before, through conditional compilation or parametrized functions, which acts as an interface for other different source files. This implies dependencies problems, if a function is made to handle different hardwares it will depend on all the specific instructions that these hardwares support. To avoid dependencies from code that is inherent to different devices respect to the targeted one, conditional compilation can solve this problem, but then managing configurations is a task that cannot be avoided, and without a proper construction, adding more elements will result in an unmaintainable code.

Five Layers Abstraction Architecture The firmware design that is done inside D.A.E.D.A.L is based on a different and specific architecture strategy in which, five abstraction layers are modeled, in order to separate code blocks in specific layers though the use of proper interfaces to decouple at most application code from the hardware dependent one.

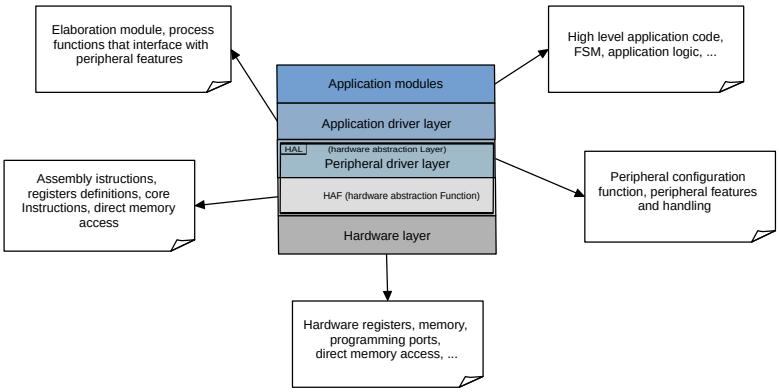


Figure 4.20: Five Layers Abstraction Pattern.

The proposed subdivision which is shown in fig. 4.20, is the following:

- Hardware: this layer consists of all the electronics that converts and actuates physical signals with the external environment through a set of dedicated memory areas, that are peripheral registers.
- HAF: hardware access function, is the part of program which grants access to hardware registers, it should only involve functions which bridge hardware registers with digital variables.
- Peripheral driver: this part groups hardware blocks that are designed to perform some specific tasks into program functions, intended to properly configure and handle them. HAF and low-level drivers compose the HAL.
- Application driver: this layer shall configure and handle peripherals through lower layers, to produce digital quantities that are totally independent from the

hardware that computes them. This layer shall be deputed to effectively decouple the application logic from the physical resources, providing an interface to the higher and lower layers.

- Application modules: this layer shall work only with the digital image of the process and the various modules shall implement the functional parts that allows the device to complete the expected application tasks.

The HAL part of this subdivision is the approach already used by micro-controller manufacturers in the development of SDKs, and it is a known approach, the innovative elements are the "application driver" and the "application modules" layers. The HAL part is independent from these two and can be done with other criteria. This is done by purpose, because application driver layer should be able to operate with all the available HALs to maintain portability. In the next sections, the functioning and construction of these two layers is presented showing what has been introduced here, and how to realize this structure and the derived benefits.

Application Driver Layer : it can be considered the firmware counterpart of the software "Data Converters Layer". The code elements and functions belonging to this layer offer methods and data interfaces with the purpose of translating hardware computations and sensing into primitive data types, that are used to update the corresponding process variables. This layer also contains the elements to translate process variables into peripherals data in order to accomplish the actuation part.

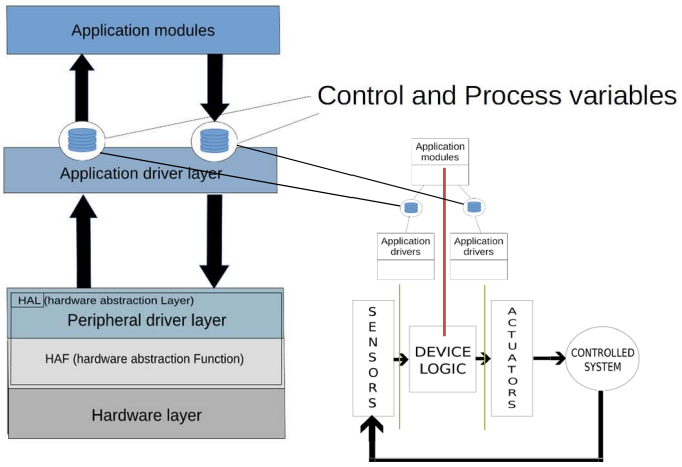


Figure 4.21: Application driver layer: decoupling application logic from hardware platform.

As shown in fig. 4.21 this layer creates a data interface with the same characteristics of the data structures presented in fig. 2.6, in the section 2.2. This data layer contains the primitive data types associated to process variables mapped in the protocol mechanisms. Just to remind the overall strategy, they have the same representation inside the whole system and they are kept coherent through the framework tools that generate these representations in a centralized way. The data encodings are not only shared through the whole framework, but as said before, they are common though all development environments, especially for C language based ones, which prim-

itive types are defined inside a proper standard. That means that application code built on these data structures, that rely only on driver methods, is totally portable, through different platforms, in which the only modifications interest the driver layer, that interface with the below HAL.

Such portability of application code is a very interesting achievement, also because, in case of technology evolution, the process logic remains the same and a verified or certified process can be left untouched, even in case of hardware improvements. These improvements could also bring the possibility of building a more sophisticated logic, but the benefit is that these two tasks can be done separately. This scenario opens different opportunities, some of them are described in the following sections. The overall strategy should be clear now, but there are two missing elements; how to do that? and how does this strategy impact on programs? To implement this strategy it is necessary to build proper interfaces between the layers and proper mechanisms to handle them. Especially for the interaction between the two upper layers, the constraint is that the only possible interface are data. It is also necessary to design and build the driver layer in the proper way. The interface construction is treated in one of the following sections, inherent to firmware construction. Design aspects of the application layer are presented after the section inherent to the design pattern proper of this structure.

4.2.2.3 Five Layers Modules Pattern

The five layers subdivision, is the enabling condition of this pattern, which is a structural one. The possibility of creating code libraries and subdivision into folders and files is something that quite every programming language does, in order to reuse entire blocks with minimum integration efforts and without modifying the code if they are complete and tested solutions.

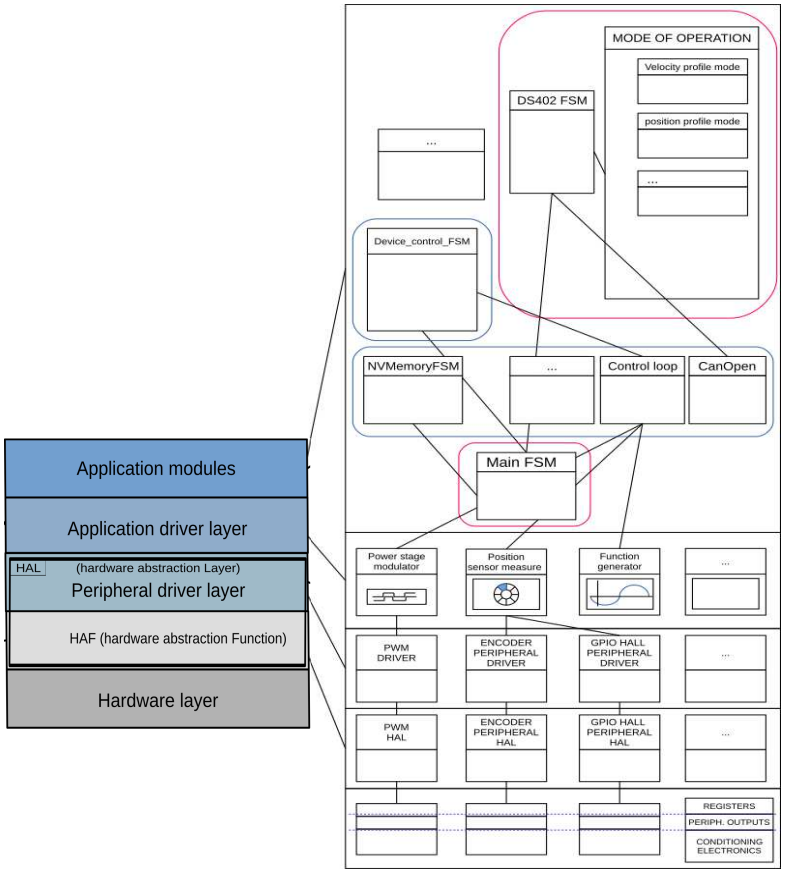


Figure 4.22: Five Layers Modules Pattern.

The concept of modularity is at the base of every engineering context. Inside programming environments, these modules can identify files and code blocks con-

structed to be fundamental element of bigger compositions. Modules often identify blocks that can be exchanged, added and removed easily, that implement specific functionalities. The proposed pattern exemplified in fig. 4.22, has been applied in particular, in the design of electric drives firmwares. But this pattern can be adopted in every "bare-metal" program. Its rules are simple:

- Functionalities shall be subdivided in portions.
- Every portion shall be contained inside a module (a library, a folder hierarchy with files), which could be easily identified and extracted from the program structure.
- Modules shall belong to only one of the proposed layers and should realize their functionalities only interfacing with modules that belong to the same or lower layers.
- The dependencies between modules shall follow the same hierarchy. For example a module belonging to application driver layer can use "include" directives only on header files of other application driver or HALs modules.
- Modules dependencies shall be contained only in the interface mechanisms.
- Application modules can interface with lower layers only exposing their data structure, which can be mapped inside communication protocols, through a proper interface construction.
- Every module shall have and activation or deactivation flag, also accessible at run-time, useful in case of faults, but also to execute the code of the necessary modules only.

The understanding of these rules should be clearer looking at the example in fig. 4.22, and also if the C file inclusion mechanism is known by the reader. This pattern responds to the needs of separating the code portions which have different hardware dependencies levels and effectively grant an abstraction from the hardware. It also enables the development of firmware functionalities based on data interfaces which are

the process variables, which can be used to build interactions between the software and firmware elements, through the shared digital image, which has been previously described. It offers a modular structure in which elements can be easily exchanged through the definition of interfaces, that will be exemplified in the construction section. The integration efforts are contained in the interfaces and do not affects modules logics. Modules logics can be functions or FSMs that are properly executed inside routines and rely on this global data structure for their synchronization, which is also their state representation. The rules that establish the modules hierarchy, are the same for dependencies and these are able to avoid the problem of circular or recursive inclusions, which can increase the build time of firmware code and sometimes they cannot be resolved, denying code compilation. The realization of C FSMs and functions, through these guidelines, should be a task that every C programmer is able to accomplish, because it involve only known programming concepts. In fig. 4.22, some of the most important modules of an electric drive are modeled. For example the control FSM relies on the control loop module, which needs drivers to calculate rotor position and a proper modulator to actuate the pulse width modulation (PWM) on the power stage. These modules then rely on specific functions to configure and access peripheral registers. In this case if a different power stage is developed, like an H-brigde instead of three phase bridge, it is sufficient to change the drivers that converts currents and voltages into data. Obviously to switch from a motor control to another, there are other aspects that have to be considered, but this example is just to show the pattern mechanism. The example of position sensor is maybe clearer; motors can have different sensors, incremental or absolute, but at the end the obtained information is an electrical angle that gives the rotor position, which is necessary to actuate the proper modulation. Changing the piece of code that writes into the position variable, totally masks the sensors type and resolution (of course different sensor can achieve different performances). For the control it is only important to know the position, not the way in which it is computed.

To give another exemplification of this pattern, in figure 4.23, it is represented the subdivision of code modules inherent to a serial communication protocol. Serial peripherals can be realized in different ways by manufactures, buffered or non

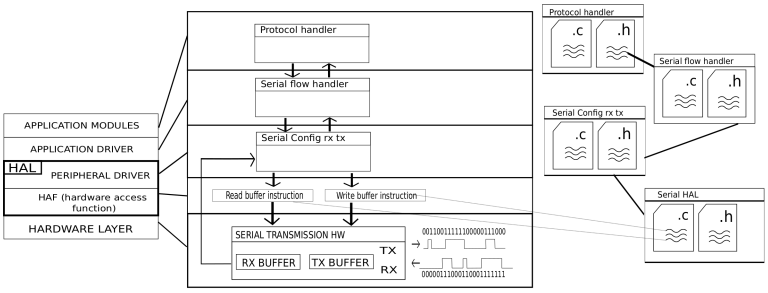


Figure 4.23: Examples of Five Layers Module Pattern usage in the design of code modules inherent to a serial communication protocol

buffered, hardware flow control, buffer sizes and other aspects. These options are handled by the proper HAL layer. At the lower layer there are the registers which can be read and written to use the peripheral. Then a upper module can be built with the proper instructions to configure the transmission in the desired way, for example interrupt driven or by polling mechanisms. In the end, every serial communication has to handle a flow of bytes, that are interpreted as messages by the protocol logic. Through a proper driver module that interface HALs with protocol logics, the same logic can be used on different devices with also different HALs without modifying protocol code parts. This also improves the reliability of the construction process, because if a protocol logic is tested and reliable, communication errors in development stages will be inherent to the wrong HAL handling or implementation. These examples are modeled without the use of standard symbols, because it is hard to describe such kind of structures with known modeling tools, and UML cannot be used due to the fact that these blocks are not classes. Usually functions inside folder or libraries are represented with header and source files. This notation has been designed with Inkscape and other vectorial graphical editors like Draw of the Libre Office. This pattern and also this notation have been used to handle team works inside industrial collaborations and have been effectively used to coordinate the development. This to remark that probably, proper methodologies for these environment should be further improved and this could be a good subject for researchers and practitioners.

4.2.2.4 Application Driver Layer Design

Application drivers design is a crucial part of this design, exploiting the module pattern, drivers can be composed as modules which interface directly with process variables and they are triggered by the modules that request the actuation or the update of variables. The design of this layer can be done in many ways, and the proposed pattern is not the only option. It could be designed also as a monolithic piece of code the handles everything or could be organized in small functional modules, such as some examples shown is fig. 4.24.

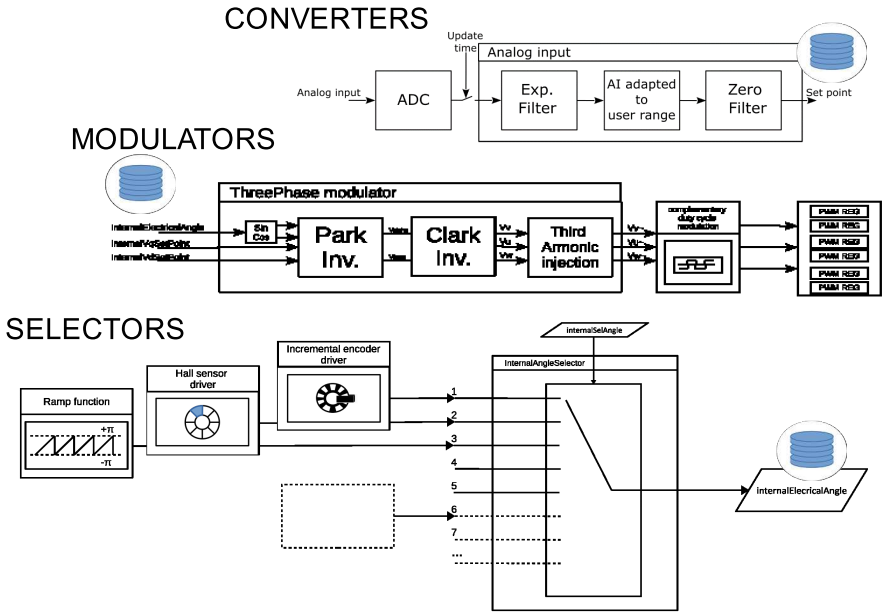


Figure 4.24: Examples of application driver layer modules.

In case of HALs that exposes data structures by themselves, the design of the relative driver is very easy, but in any case there is always the possibility of constructing a module that expose a data structure using HALs functions. Over the application driver layer, programs lose everything that can be traced back to hardware dependen-

cies.

Selectors Selectors are a special category of application drivers, the act as multiplexer, in which the index selects a module that is deputed to the update of a process variable. They can be implemented as simple use cases. Their indexes can be used also to switch sources of data during execution if the corresponding modules are enabled. This feature is useful in many cases, for example to choose proper source of position reference through the available sensors, and also to change data flows at execution time.

4.2.2.5 Modes of Operation

Till now the design has interested most of the structural aspects of the programs and their functionalities, but as told for the software, programs are not only code, but also execution flows. A generalization of the behaviour of the program of these devices has been proposed in section 4.2.2.1. In the electric drives development there is an international industrial standard known as "DS 402", from CiA consortium [88], which provides guidelines for the implementation of electric drives firmware. These guidelines define FSMs of the behaviour of the device which has to follow some stages connected though transitions that are triggered by specific commands or events, see fig. 4.25. In this figure there is the representation of the proposed state machine that should be implemented to handle the whole behaviour of the drive. According to the standards at every stage devices have to perform some task and are able to operate only in the state called "Operation Enabled".

Function	FSA states							
	Not ready to switch on	Switch on disabled	Ready to switch on	Switched on	Operation enabled	Quick stop active	Fault reaction active	Fault
Brake applied, if present	Yes	Yes	Yes	Yes	Yes/No	Yes/No	Yes/No	Yes
Low-level power applied	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
High-level power applied	Yes/No	Yes/No	Yes/No	Yes	Yes	Yes	Yes	Yes/No
Drive function enabled	No	No	No	No	Yes	Yes	Yes	No
Configuration allowed	Yes	Yes	Yes	Yes	Yes/No	Yes/No	Yes/No	Yes

Transition	Event(s)	Action(s)
0	Automatic transition after power-on or reset application	Drive device self-test and/or self initialization shall be performed.
1	Automatic transition	Communication shall be activated.
2	Shutdown command from control device or local signal	None
3	Switch on command received from control device or local signal	The high-level power shall be switched on, if possible.
4	Enable operation command received from control device or local signal	The drive function shall be enabled and all internal set-points cleared.
5	Disable operation command received from control device or local signal	The drive function shall be disabled.
6	Shutdown command received from control device or local signal	The high-level power shall be switched off, if possible.
7	Quick stop or disable voltage command from control device or local signal	None
8	Shutdown command from control device or local signal	The drive function shall be disabled, and the high-level power shall be switched off, if possible.
9	Disable voltage command from control device or local signal	The drive function shall be disabled, and the high-level power shall be switched off, if possible.
10	Disable voltage or quick stop command from control device or local signal	The high-level power shall be switched off, if possible.
11	Quick stop command from control device or local signal	The quick stop function shall be started.
12	Automatic transition when the quick stop function is completed and quick stop option code is 1, 2, 3 or 4, or disable voltage command received from control device (depends on the quick stop option code)	The drive function shall be disabled, and the high-level power shall be switched off, if possible.
Transition	**Event(s)**	**Action(s)**
13	Fault signal (see also /CiA402-3/)	The configured fault reaction function shall be executed.
14	Automatic transition	The drive function shall be disabled; the high-level power shall be switched off, if possible.
15	Fault reset command from control device or local signal	A reset of the fault condition is carried out, if no fault exists currently on the drive device; after leaving the fault state, the fault reset bit in the controlword shall be cleared by the control device.
16	Enable operation command from control device, if the quick stop option code is 5, 6, 7, or 8	The drive function shall be enabled.
It is not recommended to support transition 16.		

Table 4.1: States and transitions of DS 402 Finite State Automaton (FSA) [6].

In table 4.1, there are two images taken from the draft proposal of the DS 402, of which CiA consortium owns all the rights. Their standards are available through draft and also through membership. These pictures are useful to understand how this stan-

dardization has defined the Finite State Automaton (FSA) proper of the electric drive. In fig.4.25, there is the FSA and additional explanations inherent to the fact that the standard provides some specification of operating modes that devices can support, such as speed control, torque control, position control and many others, leaving also the possibility of extending these guidelines with manufacturers specific modes. It is common for a drive that an operating mode is selected before entering the operation enable state and then it executes the operating mode logic. This specific structure has to be implemented in order to comply with that standard which is one of most used in industrial automation for what concerns electric drives. The interesting part is that this state machine has a similar composition as the one described in fig. 4.7. In fact in one of my working experiences, acting as interface between the CiA consortium and TeMec Drive for the standardization of electric drives firmware, both FSMs were implemented inside a firmware project, as it is also modeled in the example in fig. 4.22 in which there is also the block named "DS402" FSM which relies on the main one. The DS402 could be considered as a set of specific tasks for the electric drive that the main finite state machine has to execute in addition to the necessary ones to execute the application code. This should be also an experimental proof of the capabilities of the modular design in which these FSMs can be executed and synchronized through the data which contains also their state. Anyway the standardization done by CiA consortium is very versatile and probably is one the reasons that has led to the diffusion and adoption of this standard, which helps machine producers that can use different drives exploiting their common structure and mechanism. The way in which manufacturers realize drives influences performances and feature, but if a supervisor unit needs to use position control, the interface mechanisms are the same, and for that unit the presence of a drive or another is totally masked. Obviously things are not so simple, drives need to have their own parametrization and some could support more features than others, but generally there is a reference model of what drives have to do, in special way thanks to the work of the CiA consortium and also of the others international and national Standards Organizations.

The various FSMs, see fig. 4.25, inherent to operating modes, emergency behaviour and other application logics, can be implemented using the proposed pattern.

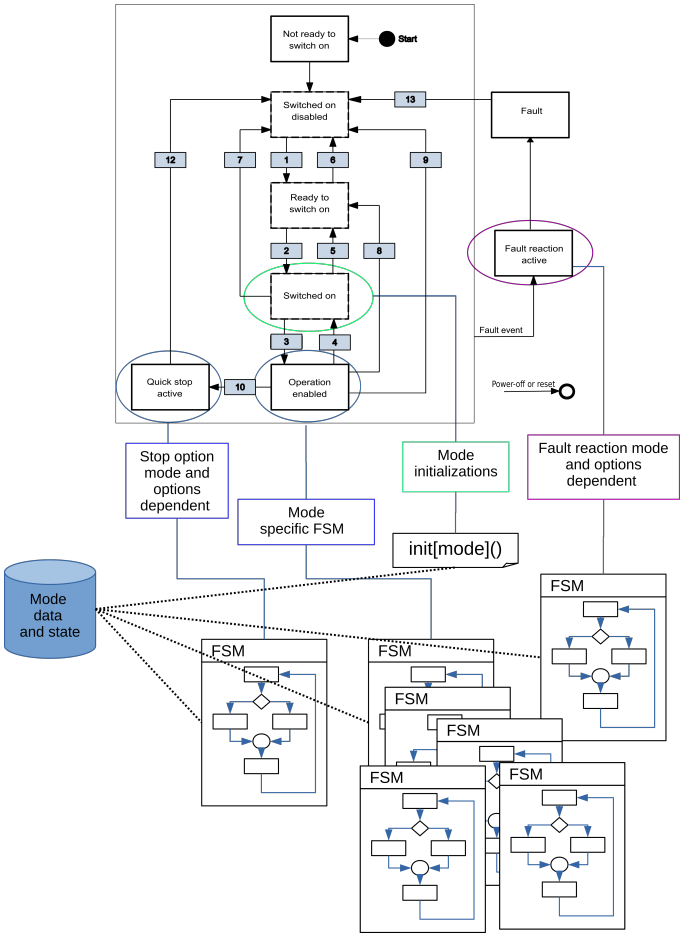


Figure 4.25: DS 402 FSA with modes FSMs.

Especially operating modes which rely on the device control loop can benefit from a module which has been designed to achieve a configurable control loop, which maximize the reuse of algorithms code. All these FSMs are related to application code, which can benefit from the proposed pattern in which it is easy to exchange modules, also at run-time, as the case of the selectors category of application drivers.

4.2.2.6 Stage Functions Module Pattern

In the whole writing about design, it has been defined what a module is, but not how these modules are designed. The only consideration about this, is the one about interfaces, in which application modules have to rely on the data ones. A generic module is composed of:

- Data.
- Methods.
- An internal state (represented through data).

Not all the modules need these three elements, especially the HALs ones which can be composed of few methods. For generic modules it could be convenient to define a behaviour that is coherent with the device and possibly organize its methods and execution accordingly to the overall behaviour. If a module is properly designed, following this proposed pattern, which is represented in fig. 4.26, the collocation of its methods can be bound to the device stages.

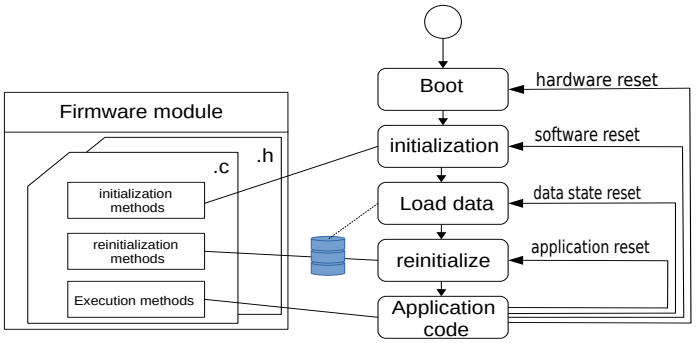


Figure 4.26: Stage Functions Module Pattern.

This subdivision aids to keep the program state coherent and also to synchronize the various execution tasks. This subdivision also enables the technique that is presented in the next section, which permits the realization of a configurable Control Loop.

4.2.2.7 XF-BAR: Configurable Control Loop

Besides the proposed patterns of this chapter, the XF-BAR is one of the most innovative concepts inside the framework. XF-BAR, is a name that derives from a concept used in some microcontroller peripherals. Freescale (now part of the NXP group), introduced a peripheral named cross-bar that permits to interconnect other peripherals with physical signals, through its configuration. XF-BAR, is a sort of firmware cross-bar that can interconnet modules that have a design proper to interface with this instrument. It is designed to execute different sequential algorithms through a program parametrization, that means the program algorithm can be modified without re-programming the device. In order to use this instrument the program that executes it has to be designed with a proper logic, as the one proposed in the previous section 4.2.2.6. The XF-BAR is composed of many elements: modules which contain low level functions and algorithms, a "for loop cycle" with parametrized executions, an array containing the variables which have to be processed, and a list of pointers to the module functions, see fig. 4.2.3.3.

XF-BAR is a technique, which is implemented inside a homonymous module. Functions of modules, which are useful for control loops, such as P.I.D controllers, additions, subtractions and every other inherent to the control process, can be added inside the XF-BAR module defining a method that encapsulates the module's one and acts as interface with an array of data, that is accessed though reference. The XF-BAR methods do not return any values but the result of the elaboration is put inside a specific position in the data array, which has to be updated with the process variables before executing the control loop. Every method can have different inputs and outputs depending on the provided execution. Output positions are fixed, instead the input indexes are configurable (they use the same mechanism as process variables). These XF-BAR methods can be used to obtain more than one operation, allocating necessary outputs in the array. For example P.I.D methods take error signals as input and produce an output, this output has a fixed position in the array, a successive invocation of the same method will overwrite the result. For that reason, more outputs can be assigned inside the vector that refer to a different P.I.D outputs. Each P.I.D output needs then to be associated with a proper structure in which there is the index of the

corresponding input. The same concept can be applied for every method, in which every execution block has its own configurable data structure. The number of blocks depends on the need of the control loops that the device is expected to execute, and they can be easily added extending the array and declaring more data structures.

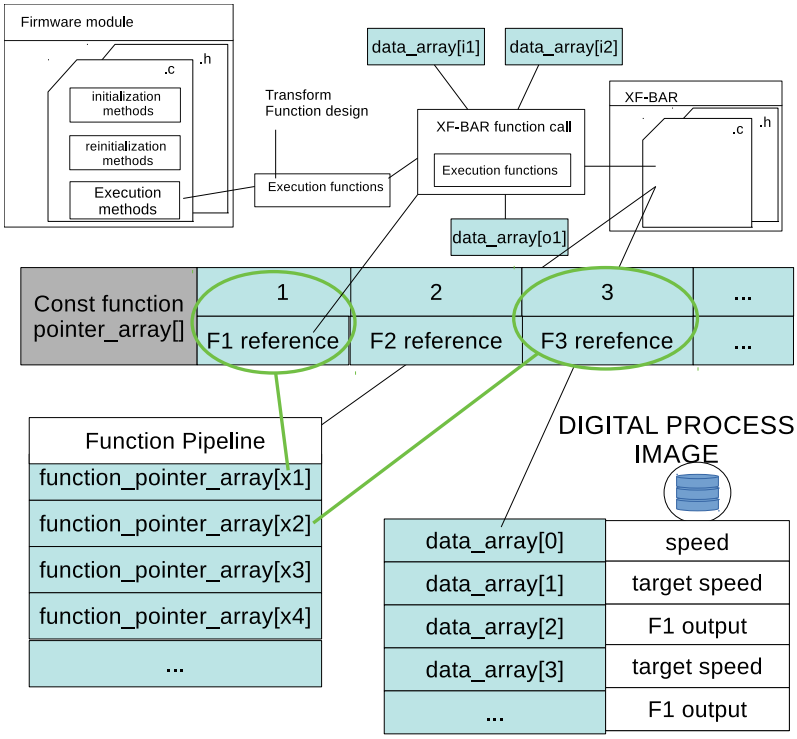


Figure 4.27: XF-BAR: Configurable Control Loop.

Every block is accessed through a specific method call, for example, "executePid1(index i)" and "executePid2(index i)". For each method the pointer reference is taken and inserted inside a **const** (constant keyword in C language) array of function pointers. Through a proper construction it is possible to configure an array which contains these function pointers of the const array, that can be called inside a code-loop through a proper syntax. This array is the "functions pipeline", which indexes are inherent to methods that have to be invoked. These pipelines, due to the static usage of

memory, have a fixed maximum size that has to be chosen to contain all the possible functions invocations of the longest loop. The loop has to be stopped through the invocation of a proper routine that states that the loop has to finish at that index. More than one pipeline can be composed, but for now the sequential execution of a single control loop is the only encountered scenario. To sum up what has been presented in this section, a brief summary is done; XF-BAR has a data array in which all the variables are copied before starting the control loop, these variables are manipulated through XF-BAR methods using their references, during the pipeline execution. The outputs of the methods are put inside the array, and can be used as input for other methods. Inputs and functions execution order can be parametrized through some indexes, which are part of the process variables, that are part of the shared digital image. **Execution flow itself becomes part of the "digital process image" and can be also manipulated**. To the reader is left the thinking of the possible applications that this kind of modules can have inside the D.A.E.D.A.L framework, which design is specific to handle this kind of interaction based on the shared digital image. Through these modules it has been possible to compose the known control loops inherent to DS 402 specifications and many other controls that can be done through a cascade of functions as the examples in fig.4.29 and in fig. 4.28.

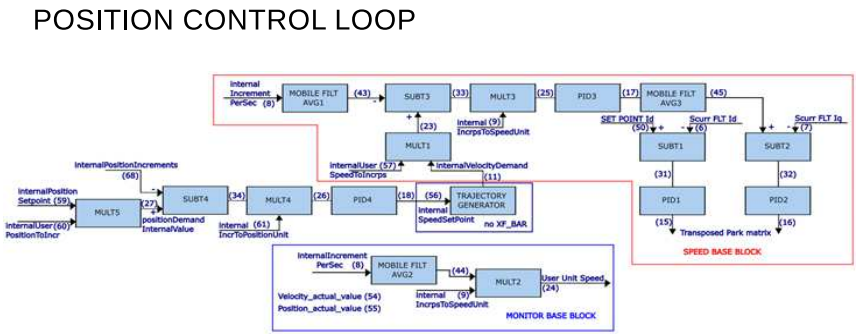


Figure 4.28: Composition of the position control loop through XF-BAR configuration.

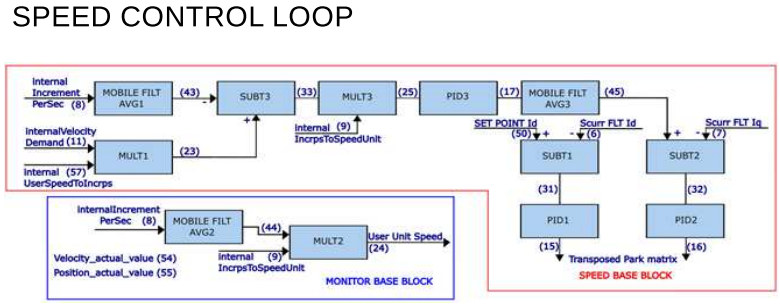


Figure 4.29: Composition of the speed control loop through XF-BAR configuration.

4.2.3 Firmware Construction

Firmware construction, as the software one, will explore only the details that can be enabling for the design strategies proposed, in order to avoid the excessive exemplification of tasks that are expected to be known by every programmer.

4.2.3.1 FSM Construction

The construction of finite state machines in C language is a well known and applied practice. They are often based on a method containing a "switch case" statement. This method has then to be called inside a cyclic routine and its state changes when the conditions for a state transition are verified. Fig. 4.30, there a simple example of finite state machine construction.

```
void device_fsm()
{
switch (state_device_FSM)
{
case DEVICE_FSM_IDLE:
state_device_FSM=DEVICE_FSM_READ_MEMEMPTY;
break;

case DEVICE_FSM_READ_MEMEMPTY:
      ...
break;


case DEVICE_FSM_REINIT_DATA:
       if(requestNVM==noRequestNVM)
       {
              if(crcErrorPeristenceNVM==1)
              {
                     state_device_FSM=DEVICE_FSM_LOAD_ERROR;
              }
              else
              {
                           //reInitDataAtReset
                     reInitDataAtReset(&device_taurus_sm);
                     reinitPeripepheralOnReset();
                     reinitApplicationDriverDataAtReset();
                     reinit_application_module_data_at_reset();
                    state_device_FSM=DEVICE_FSM_START_CONTROL_LOOP;
       }
              }
       break;

case DEVICE_FSM_MEMORY_ERROR:
       state_device_FSM=DEVICE_FSM_START_CONTROL_LOOP;
       break;

case DEVICE_FSM_START_CONTROL_LOOP:

       …

}
```

```
while (1U)
{
  //START UP & low priority operation
                          device FSM
  device_fsm();

  NVMemoryFSM(&device_data);
  I2C_DriverFSM();
  EEPROM_DriverFSM();

  …
 }
```

Figure 4.30: FSM construction example in C code.

4.2.3.2 Modules Interfaces

Interface construction should be also a known activity derived from the language capabilities, but due to the fact that this specific aspect is an essential part of the design it is important that a proper exemplification is provided.

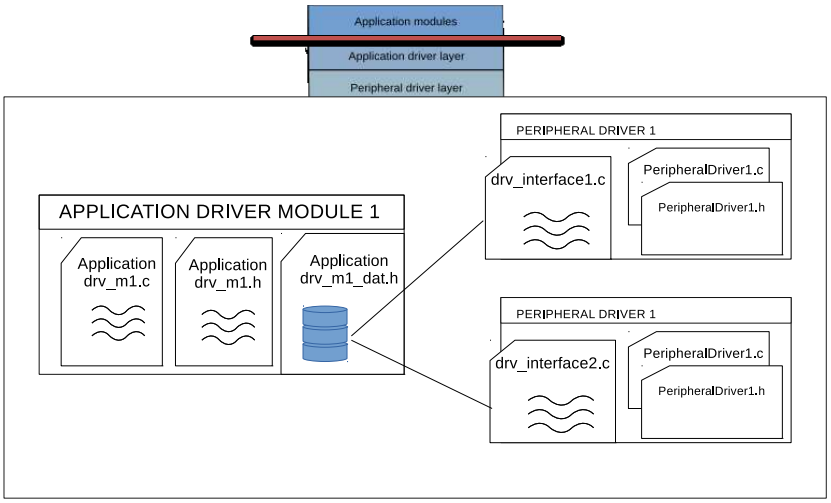


Figure 4.31: Firmware modules data based interfaces.

In fig. 4.31 it is modeled the file composition of modules based on data interfaces, as stated before this kind of interface have to be adopted by application drivers that interface with application modules. It can be also used in other contexts and its use in the proposed pattern is what contributes to the building of the process variables. The interface is implemented in a file in which the data structure is contained and data are accessible by inclusion of the header file by other modules or externalization of the data structure through the **extern** instruction. The same header is accessed by module files that operate with data in it. Externalization makes variables visible at global scope, and they are then accessible from all the interested modules. This op-

tion totally cuts dependencies, but modules has to handle the accesses to resources to avoid incorrect handling of data (this is often easy due to the fact that execution is almost always single-threaded inside micro-controllers). The other option is to include the data header file inside other modules. This option can build more sophisticated interfaces which implement also methods for interfacing, but creates dependencies between modules.

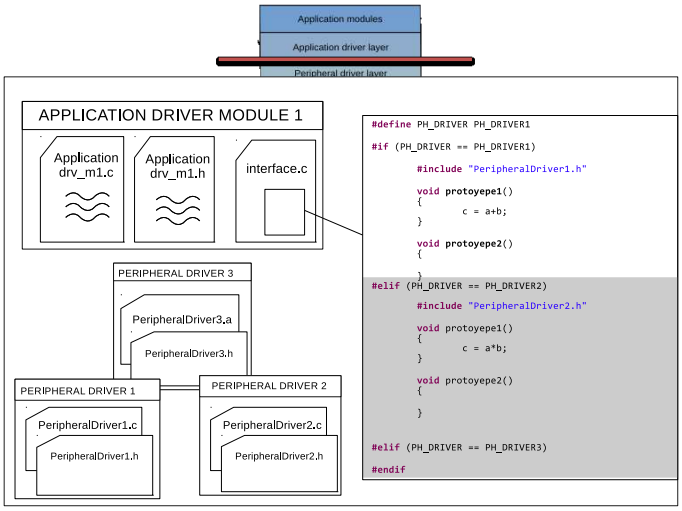


Figure 4.32: Firmware modules prototype based interfaces.

In fig. 4.32 it is modeled and exemplified the interface prototype mechanism with conditional compilation. This kind of interface is very useful for the interfacing of lower layers in which there is often the need to write different instructions and rely on different libraries. Through conditional compilation it is possible to define the prototypes and include files only when needed. This option allows to handle and separate dependencies and implement specific methods.

All these techniques are based on known programming mechanisms such as the

```c
output_return_t retunBasedMethod(input_param_t input,
        input_param_reference_t inputReference)
{
        if(error_condition != 0)
        {

                return errorCode[error_condition];

        }


        output_return_t output; //declared here only
                                //for example purposes


        return output; //last line of the method
}
```

```c
void trasnformationBasedMethod(
        s_input_struct_t *inputReference,
        s_output_struct_t *outputReference )
{

        //elaboration of input and output data
        outputReference->errorCode[error_condition];

        outputReference->output = inputReference->input1+
                inputReference->input2+
                        outputReference->memory;

}
```

Figure 4.33: C code examples: return based method function, on the left, transformation based function, on the right.

different ways in which methods can be implemented, see fig. 4.33, like functions that receive parameters by value or reference and return a value. Another possible implementation is the "transformation" function, which does not return values but only manipulates data through their reference. Except for the application of these programming techniques in the realization of the interfaces between modules, which can be seen as an application of them into a new context, the presented construction mechanisms are proper of the language.

4.2.3.3 XF-BAR

The construction of this module is something that is always based on language features, but their composition is used to realize a complex structure which is something that can be considered an innovative approach, especially when combined to the digital image mechanisms of D.A.E.D.A.L, which realize the parametrized execution flow described in the design section 4.2.2.7. The construction of this mechanism could be nontrivial also for C language experts and its exemplification can also aid the understanding of its functioning. Fig. 4.34 shows the pipeline execution through

```
void xf_process_pipeline()
{
   pipeExecCounter =0;
   pipelineEnd = PIPELINE_IN_PROGRESS;

   while(pipelineEnd != PIPELINE_END)
   {
      xF_pipeline_function_array[pipeExecCounter]();
      pipeExecCounter++;
   }
}
```

```
//signal ending of pipeline
void xFbarPipeLineEnd()
{
      pipelineEnd = PIPELINE_END;
};
```

Figure 4.34: XF-BAR: C code construction of execution pipeline.

the conditioned loop and the termination function that stops the pipe processing. As can be seen the syntax of methods invocation inside the loop is a particular one in which round brackets "()", are put at the end of an array. This array contains function pointers that can invoke routines through this particular syntax.

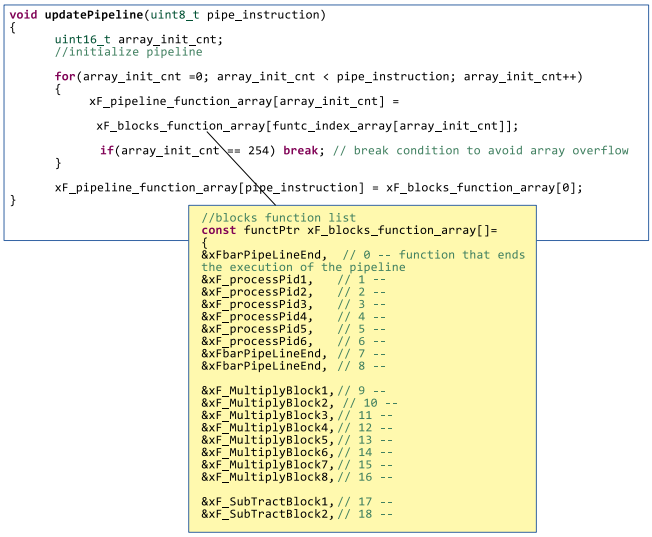


Figure 4.35: XF-BAR: C code construction of pipeline configuration method.

Fig. 4.35 shows how the pipeline instructions are put inside the array through the const array containing functions references and how that array is constructed. This process is always done through the configuration of an array of indexes which choose which functions to invoke and their order.

```
void xF_MultiplyBlock(uint8_t blockN)
{
      xF_multiplierBlocks[blockN].out =
     (xF_inputs_array[xF_multiplierBlocks[blockN].i1]) *
     (xF_inputs_array[xF_multiplierBlocks[blockN].i2]);

          (xF_inputs_array[blockN + XF_MULT_OFFS]) =
     xF_multiplierBlocks[blockN].out;
};
```

```
    void xF_pidBlockProcess1_8(uint8_t PIDn
    {
        if(pidProcessEnabled==1)
        {

              pid_process(&xF_pidParamsStructArray[PIDn],
              &xF_pidDataStructArray[PIDn]);

              xF_inputs_array[PID1_8_offset + PIDn]) =
              xF_pidDataStructArray[PIDn].OUT);
        }
        else
        {
             (xF_inputs_array[PID1_8_offset + PIDn]) = 0;
        }

  };
```

```
  void xF_processPid1()
  {
        xF_pidDataStructArray[0].IN =  (xF_inputs_array[xF_muxPid1]);
        xF_pidBlockProcess1_8(0);
  };
```

Figure 4.36: XF-BAR: examples of C code construction of XF-BAR methods.

Fig. 4.36 shows some implementations of XF-BAR methods with the characteristics that are described in the XF-Bar design. Every method accesses a data array through proper indexes that are inherent to blocks configurations. Fig. 4.37 shows the

```
//xF_inputs_array signals filling
xF_inputs_array[1] = (sCurrIuIvIw.fltA);
xF_inputs_array[2] = (sCurrIuIvIw.fltB);
xF_inputs_array[3] = (sCurrIuIvIw.fltC);
xF_inputs_array[4] = (sAlphaBeta.fltAlpha);
xF_inputs_array[5] = (sAlphaBeta.fltBeta);
xF_inputs_array[6] = (sCurrIdIq.fltD);
xF_inputs_array[7] = (sCurrIdIq.fltQ);
xF_inputs_array[8] = internalIncrementPerSec;
xF_inputs_array[9] = internalIncrpsToSpeedUnit;
xF_inputs_array[10] = (s_drive_data.s_enc0_data->electrical_angle);
```

Figure 4.37: XF-BAR: C code example of data array variables insertion.

insertion of variables into the data array which has to be an array which type is the biggest in size, in order to be able to contain all data types. This often requires more memory than would be needed to contain these data, but the difference is of some bytes only and it is the only way to handle all the data thought the same structure. Inside the array are needed only the variables inherent to control loop process.

4.3 Case study: "traditional" cascade control loop implementation and XF-BAR implementation.

Among firmware results, a noteworthy one, is the XF-BAR 4.2.3.3, that drastically reduces development time. It is a clear example of application of the proposed method. The electric drive cascade control loop, is itself part of the "digital image" and through the handling of these data, it is possible to manage the control loop, in order to obtain the desired process.

The implementation of the cascade control requires to: acquire data from sensor, generate the error data, which is the difference between the measured quantity and a given reference, processing it with a regulator, which often is a P.I.D one, and then actuating the regulator output though the proper hardware peripheral. The most common metrics, like execution time, memory occupation, lines of code are not the right instruments to evaluate this subject, as instead, can be for the works presented in annex A. In this case modularity, flexibility or other criteria proper of the software quality should be addressed, but most of the time are evaluated only by certification entities, that evaluate the product mainly considering the field of application. As stated in the whole writing, this topic is facing a drastic evolution and for that reason is hard to find comparison and proper metrics in the academic environment, even if the importance of these studies and the possible resulting benefits are clear. As happened for the software, the empirical methods and on field experience, often cover an important role in these subjects, and case studies are often the way of proposing and discussing these kind of solutions.

"Traditional" approach: This paragraph presents a comparison between controls realized in the way in which programming is usually done and through the use of the XF-BAR. The listed examples are taken from real projects which are currently part of industrial products. Fig. 4.38 and fig. 4.39, shows examples of functioning implementation of cascade controls taken from some projects which I used to work at the beginning of my PhD. In the proposed examples there are common lines of code that can be grouped in common functions and of course, some more optimization

could be done, but it can be noticed that every control loop needs the declaration of its own regulator, error calculation and the routing of quantities (red boxes in figures). Even trying to reuse common code, when there is the need to change controls or options, the whole code structure has to be modified. In addition to that, in control loops and measures, there are often filters or factors that add options to the whole control. Devices with few types of controls and few options are not hard to maintain and their flow can be still quite easily understood, but when there are many options and different kind of controls, functions can grow in number of lines of code as shown in the proposed figures in which the simple addition of the current control to the speed loop adds a lot of code and implies a different routing of the involved quantities. To make another example the composition of a position control loop usually uses the cascade of three loops; current, speed and then position, that would imply an additional regulator in the proposed example. A simple torque control uses the current control loop with a factor that is the torque constant of the motor. Handling only these few examples, even trying to maximize the reuse of common lines of code, it could be a time consuming task for development, testing and maintenance of the program code.

D.A.E.D.A.L approach: Exploiting the XF-BAR and application driver layer, as shown in the previous chapters, data like speed and current measurements are stored inside an array of digital values that can be manipulated during execution.

Fig. 4.40 shows and example of implementation with the proposed approach, these are more or less the same common lines that are needed also in the traditional approach, plus some more "selectors" cases, that do not add more lines to the needed feature, of course more cases imply more features and more lines, but for the single one, as the routing of values, it requires few instructions in both cases. The common lines are only organized in a way that promote flexibility and modularity. For control loops instead it can be seen that adding filters or loops requires the writing and testing of the method in which the control is coded. Every option needs to be coded as a function, in which common portions of code can also be shared, but usually it is required the declaration of some variables and the construction of the proper instruction sequence inside the method. Using the XF-BAR feature, the composition of

```
void simpleSpeedLoop()
{                                                          SPEED MEASURE
//speed measure
speedpos_update(enc_counter, &encoder0_data);

//measure filtering
mobile_avg_filter_f(&speedFreqFilter, encoder0_data.radpm);

                                                   SPEED CONTROL LOOP
//error calculation
errSpeed = radPmSetPoint -  speedFreqFilter.media;

//P.I.D regulator processing
speedPidData.IN = errSpeed;
pid_process(&speedPidParams, &speedPidData);

                                                   OUTPUT ACTUATION
//output actuation through three phase modulation for brushless
motor
vqstar = speedPidData.OUT;
vdstar = 0;

pdQ.fltArg1 =vdstar;   //id
pdQ.fltArg2 =vqstar; //iq

GMCLIB_ParkInv_FLT(&pAlphaBeta, &pAngle, &pdQ);
    GMCLIB_ClarkInv_FLT(&pUuVvWw, &pAlphaBeta);

Vu = (pUuVvWw.fltArg1 );
Vv = (pUuVvWw.fltArg2 );
Vw = (pUuVvWw.fltArg3 );+

dutyCycleA = (uint16_t)(Vu*voltToPwmPeriod +50);
dutyCycleB = (uint16_t)(Vv*voltToPwmPeriod +50);
dutyCycleC = (uint16_t)(Vw*voltToPwmPeriod +50);

//to pwm pheripheral
center_aligned_3ph_pwm(dutyCycleA, dutyCycleB, dutyCycleC);
}
```

Figure 4.38: Firmware code example of "traditional" implementation of speed control loop

```
void currentControlledControlLoop()
{                                                          CURRENT MEASURE
//current sensing and conversion to id and iq axis
currADC[0] =  -(float)( (int32_t)(((int32_t)*chA) -
(int32_t)channelAdcOffset[0]))*ADC_PHASE_CONV_FACTOR;
currADC[1] =  -(float)( (int32_t)(((int32_t)*chB) -
(int32_t)channelAdcOffset[1]))*ADC_PHASE_CONV_FACTOR;
currADC[2]= - (currADC[0]+currADC[1]);

pCurrUVW.fltArg1 = currADC[0];
pCurrUVW.fltArg2 = currADC[1];
pCurrUVW.fltArg3 = currADC[2];

pAngle.fltArg1 = GFLIB_Sin_FLT(angleRad, GFLIB_SIN_DEFAULT_FLT) ;
pAngle.fltArg2 = GFLIB_Cos_FLT(angleRad, GFLIB_COS_DEFAULT_FLT)  ;

GMCLIB_ParkInv_FLT(&pAlphaBeta, &pAngle, &pdQ);
GMCLIB_ClarkInv_FLT(&pUuVvWw, &pAlphaBeta);
                                                           SPEED MEASURE
//speed measure
speedpos_update(enc_counter, &encoder0_data);

//measure filtering
mobile_avg_filter_f(&speedFreqFilter, encoder0_data.radpm);
                                                           SPEED COTNROL LOOP
//speed error calculation
errSpeed = radPmSetPoint - speedFreqFilter.media;

//speed loop
speedPidData.IN = errSpeed;
pid_process(&speedPidParams, &speedPidData);
                                                           CURRENT CONTROL LOOP
//current error calculation
errIq = speedPidData.OUT - pCurrDQ.fltArg2;
errId = 0.0f - pCurrDQ.fltArg1;

//iq control loop
IqpidData.IN = errIq;
pid_process(&IpidParams, &IqpidData);

//id control loop
IdpidData.IN = errId;
pid_process(&IdpidParams, &IdpidData);
                                                           OUTPUT ACTUATION
//output actuation through three phase modulation for brushless
motor
vqstar = IqpidData.OUT;
vdstar = IdpidData.OUT;

pdQ.fltArg1 =vdstar;   //id
pdQ.fltArg2 =vqstar; //iq

GMCLIB_ParkInv_FLT(&pAlphaBeta, &pAngle, &pdQ);
    GMCLIB_ClarkInv_FLT(&pUuVvWw, &pAlphaBeta);

Vu = (pUuVvWw.fltArg1 );
Vv = (pUuVvWw.fltArg2 );
Vw = (pUuVvWw.fltArg3 );+

dutyCycleA = (uint16_t)(Vu*voltToPwmPeriod +50);
dutyCycleB = (uint16_t)(Vv*voltToPwmPeriod +50);
dutyCycleC = (uint16_t)(Vw*voltToPwmPeriod +50);

center_aligned_3ph_pwm(dutyCycleA, dutyCycleB, dutyCycleC);
}
```

Figure 4.39: Firmware code example of "traditional" implementation of speed control loop with current control loops

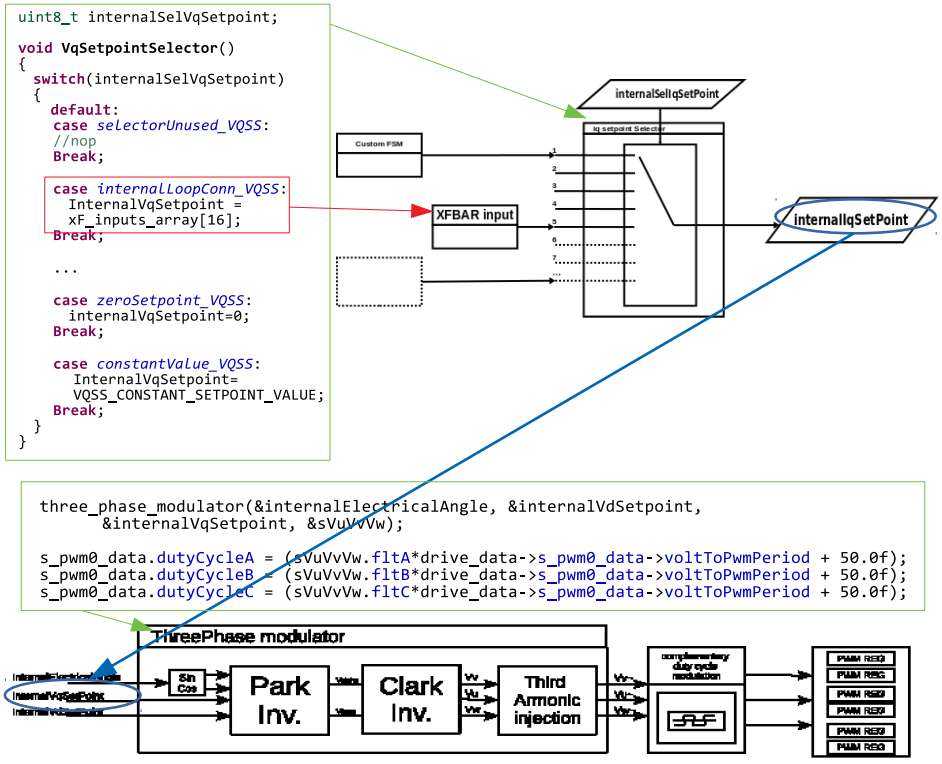


Figure 4.40: Firmware code examples of selector and modulator modules

different control loops, can be realized only assigning modules inputs and filling the function pipeline using indexes. Most of the modules have at least two inputs, so **in order to route the control loop and invoke a functionality only three lines of code are necessary at most**. This kind of solution is very effective with cascade control loops due the similarity to a functional block diagram, which in this case can also be used as reference to understand the code logic. As shown in the example in fig. 4.41, it is easy to compose a control loop using indexes and trace it with a block diagram. In the same way the different controls that are cited in section 4.2.2.7 can be composed, which are some of the actually used and developed controls inside one of the industrial projects I'm working on, that is already a commercial product. With this approach, controls can be realized through the use of tested functions and filters with few lines of code. Controls can be coded in the program logic and also, exploiting the data exchange between software and firmware, there is the possibility of configuring them through this interaction.

```
void xF_bar_current_loop()
{
  xF_subtractBlocks[0].i1 = 50;
  xF_subtractBlocks[0].i2 = 6;
  xF_subtractBlocks[1].i1 =  51;
  xF_subtractBlocks[1].i2 =  7;
  xF_muxPid2 = xF_INDEX_SUBT2_OUT;
  xF_muxPid1 = xF_INDEX_SUBT1_OUT;
}

void xF_current_loop_init()
{
   xF_bar_monitor_loop();
   xF_bar_current_loop();

   funtc_index_array[0] = xF_FUNCTION_CALL_SUBT2;
   funtc_index_array[1] = xF_FUNCTION_CALL_SUBT1;
   funtc_index_array[2] = xF_FUNCTION_CALL_PID2;
   funtc_index_array[3] = xF_FUNCTION_CALL_PID1;
   funtc_index_array[4] = xF_FUNCTION_CALL_AVG2;
   funtc_index_array[5] = xF_FUNCTION_CALL_MULT2;

   updatePipeline(6);
}
```

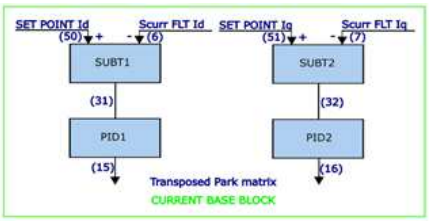


Figure 4.41: Firmware code examples of control loops with XF-BAR

Chapter 5

Results

There are many valuable results that have been obtained through the proposed framework, both for firmware and software. In my opinion the actual used metrics in software and firmware engineering are not enough to fully evaluate design solutions and approaches. When talking about algorithms, execution time, lines of codes, memory occupation, number of assembly instructions and other quantifiable metrics, these criteria, are able to denote important aspects of the comparison between solutions, but are not enough to characterize the whole design and realization of programs. In order to deal with programs, it is necessary to consider factors that not always can be modeled with the actual mathematical formalisms and neither with the actual mathematical logic. There are many factor that define the quality of a software (or firmware) program, many of them, well categorized in the corresponding subject; like reuse of code, reliability, modularity of the solution and effectiveness of the proposed abstraction, for which it is not easy to quantify the impact that they have in these fields, even if it is very resented. There are some proposed methods to evaluate these characteristics, but is still hard to find a common, simple and accepted method to evaluate them, and to which abstraction level they are evaluated, the whole design or a software component [89], [90], [91], [92], [93]. In software engineering these concepts are more accepted, while for firmware there is still some reluctance. The evaluation of these characteristics, is often done through surveys, due to the huge dependency

of the human factor involved. Unluckily doing these kind of surveys is often difficult, especially for industrial projects, in which some confidentiality is needed.

In the end, the essential concept is always to **reduce development time** and **increase reliability**, through reuse of code and structured design and programming. As already reminded programs are not only algorithms with numbers and time constraints, they are logic processes, so they have also to be evaluated with metrics that do not consider only performances.

In the analysis of the firmware results it is important to consider al least three aspects: the program architecture, the execution flow, in terms of performances and its evolution of logic states, and also the concept of abstraction. These aspects have different characterization and weights for the software part, in which there are also other implications that derive from the field of application, the user interaction and many other important facets.

5.1 Firmware results

There are different proposed methods in this work and all of them respond to the same need, facing different recurrent problems. Some of them are the enabling mechanisms of the proposed approach, thanks to the features that they are able to grant. This analysis will sum up these techniques, pointing out their characteristics which aid to achieve the concept explained before. These properties can be easily verified implementing any program logic, following the rules of the proposed patterns. Different implementations can be ported on another platform, except for layers and modules that depend on them, and can be executed in the same way once the hardware dependencies are correctly handled, remembering that they are not part of the application logic. The result of this solution is an effective modular organization of the code, the concrete possibility of having modules that can be used in CBD approaches and an effective abstraction (from the hardware) of the process logic.

5.1.1 Firmware results summary

For the above considerations the analysis of the results will interest metrics that are still hard to handle but they are recognized as key factor in program development, which are:

- Abstraction.
- Modularity.
- Flexibility.

Proposed construction patterns for "C" language:

Construction Pattern	**Abstraction**	**Modularity**	**Flexibility**
Data based interfaces	Uses primitive data types which are common to all computing systems	• Common to all systems • Decoupling through data is total	Any modifications impact only on the data structure. Data can be easily extended, modified and accessed
Methods prototypes based interfaces	Methods are defined at compilation time	Methods implementation can be easily changed through conditional compilation	Changes are limited to interface source files

Table 5.1: Construction patterns results summary

Proposed architectural patterns:

Pattern/Method	Abstraction	Modularity	Flexibility
Device State Machines	Common description and implementation for all devices related to their functioning	Every stage is independent and self confined	The structure does not need modifications to be used
Five Layers Module Pattern	• Decoupling from hardware dependency • Decoupling from application and device logic • Functional modules description	Interfaces: • Data based • Methods prototype based Modules are confined inside abstraction layers and cover specific function	• Application modules do not need modification to be used (data interfaces do not need adaptation in different devices) • Methods prototype interfaces limit changes in defined portions of code • Changes inside a module are totally masked to other modules
Stage Functions Modules Pattern	Provides modules abstraction from functional and behavioral perspective	Module functions	Changes are limited to execution stages of the module
XF-BAR	Configurable control loop: • Same composition for diffrent controls • Same fundamental blocks	Control can be composed using atomic modules that implements functions and filters	It can potentially implement every kind of execution flow, if there are no modules which implement the desired function they can be easily implemented and integrated with the other without affecting the whole functioning and the other components.

Table 5.2: Architectural patterns results summary

5.2 Software results

The main achievements for the software part are essentially two:

- The automatic generation of program applications with GUI and commissioning capabilities through industrial communication protocols (S.I.E.G.Fr.I.E.D).
- The abstraction of processes logics from the communication and implementation ones, the "Digital image" masks the implementation complexity and all the elements that are not inherent to process logics.

These two results are able to drastically reduce development time and especially in the first case increase the reliability through the use of tested code in which no additional modifications to the source code are needed. Composing software through the C.I.T editor does not add lines of code, so it is impossible for tested modules to be modified with faulty lines of code. The second result is also able to increase reliability, but this is the case in which the whole framework is used as a middleware and collection of APIs, on which a developer can build applications. This allows to reuse the common tested communication stacks and data presentation layer, so the developer can focus all the efforts on the application. This approach has proven to be very effective also in real projects in which a complete solution has been developed in few weeks, even by a developer with no experience on industrial communication protocols, as can be seen, by real application on the field, from sections B.1 and B.2.

5.2.1 Software results summary

The proposed innovative software patterns, which have permitted to reach these results, will be summarized in this section, pointing out their characteristics.

Architectural Pattern	**Abstraction**	**Modularity**	**Flexibility**
Editor-Generator-Data-Modules	C-b-C: software can be constructed by configuration, not programmed	CBD, software applications are designed using atomic component, both graphical and logical	Adding a feature implies only the addition of a case statement in factory classes. Adding a module does not need changes in the existing ones.
Factory Propagation Pattern	Effective implementation of OO concepts: • Object composition and encapsulation • Inheritance	Program objects are created in accordance with the needs through defined classes as the traditional factory pattern	Factory patterns have been created with the purpose of achieving high degrees of flexibility • objects are created in case of need without prior knowledge (easy to adapt) • Adding new objects implies modification only in factory cases.
Data Converters Pattern	Effective implementation of OO concepts: • Object composition and encapsulation • Inheritance • Polymorphisms **Creates the process variable (part of digital image) abstraction**	• Every process variable can be interfaced with more graphical components of any kind • Every process variable can be referred to any set of protocol converters • Every process variable and its referenced elements are independent form the others, even if they are kept coherent with the digital image	Modifications of the program presentation or communication protocol can be easily done only adding or removing object references to the existing converter layer, through dedicated interfaces without modifying any lines of code of the converter layer.

Table 5.3: Software architectural patterns results summary

5.2.2 Final result: D.A.E.D.A.L Framework.

All the obtained results were finalized to the realization of the components of the whole framework, that has been realized exploiting all the proposed techniques which have been translated into code libraries, projects and programs, that are actually used in industrial environments.

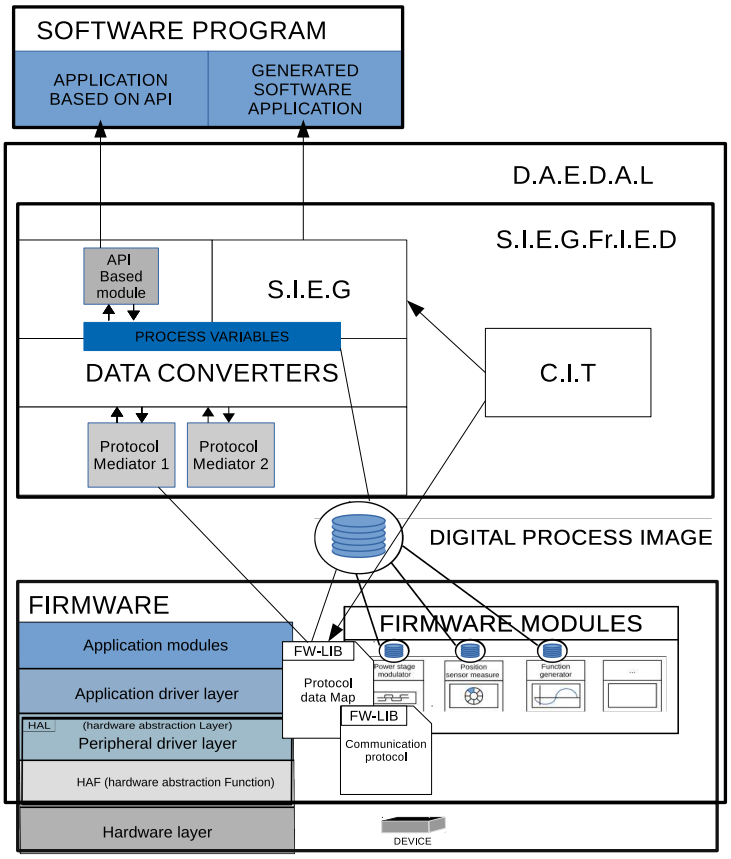


Figure 5.1: D.A.E.D.A.L DIAGRAM REPRESENTATION

5.2.3 D.A.E.D.A.L results summary

The realization of the S.I.G.Fr.I.E.D software framework and innovative firmware design techniques brought enormous advantages, in terms of reduction of development time and increase of reliability. Protocols and data maps handling, variables and registers conversion are very repetitive tasks, in which it is very easy to make mistakes, most of them attributable to indexes misspelling and wrong messages content reconstruction into variables and this operation has to be done both for the software and firmware data. Also, for software graphical interfaces, the creation of panels for data input, manipulation and parametrization often implies the repetition of the same graphical components in which the variable that has to be properly handled and manipulated is linked. This approach replaces the coding activities that are necessary to accomplish these tasks, with computer aided design operations in which no coding is needed and the validity of the data input is checked by the editor itself.

General results: Abstraction, modularity and flexibility, are recognized as key factors to reduce efforts in adapting programs when requirements change or simply in their evolution, in order to offer more services. Through these properties code reuse is maximized and enables a CBD approach, with the consequent benefits, like happens in other fields of engineering in which the composition of complex systems is done assembling different components. An high level of abstraction, like the "process digital image", grants a deeper understanding of the programs behavior and interactions.

S.I.E.G.Fr.I.E.D. results: The editor program allows to generate a synchronized data structure for the firmware data maps and the automatically generated commissioning software program. This allows to handle the configuration management of these two elements. Every generated files keeps trace of the data variables that are present in a particular version of the program. Through this data, the software can also understand which version of firmware is present on the device (the generated data maps contain identifiers for versioning purposes). Usually software and firmware are provided with compatibility matrix that inform the user which version of software can be used with a particular firmware version. Through this solution the software is able

to be totally backward compatible with previous firmware releases. Summarizing the achieved results:

- The editor software is able to generate the complete commissioning software that can exchange data with any device that support the implemented protocols.
- The editor can provide the source code of the firmware data maps of the communication protocol stacks.
- If the editor generated source code is used, it is granted the data-coherence between the device and the commissioning tool.
- The commissioning software and the protocols maps, are configured, not coded, using a C-b-C approach. This implies:
  - Configuration is a lot easier and faster than programming, there is no need to know program languages or communication protocols mechanisms, at least to produce a commissioning software.
  - Inside generated programs it is also possible to code other modules without conflicts. So programming and Configuration can coexist, meaning that **there are no restrictions derived from this approach**.
  - Configuration management is handled automatically for the most.
  - When configuration mechanisms are tested, there is no possibility to insert faults in the generated applications, meaning that **robustness and reuse of code are maximized**
  - Communication is reliable, because it is built on the same tested layer.
  - Data coherence is granted for both firmware and software, that share the same abstraction of the process.

Chapter 6

Conclusions

This work is the result of many years of researches and some experience in the industry, that started even before the PhD. This result is the realization and application of the D.A.E.D.A.L framework, in the development of the virtual components of industrial embedded devices. D.A.E.D.A.L framework is the composition of its elements and patterns, that define its strategies. D.A.E.D.A.L framework faces the development of virtual products, both for the software and firmware components. The software part of D.A.E.D.A.L, Development Aid for Embedded Devices Automatic Labor, is the software framework S.I.E.G.Fr.I.E.D, Software Interface Environment Generation Framework for Integration of Embedded Devices, which has matured in complex software programs that are able to automate the development of softwares with GUIs, and also generate firmware source code for protocol data maps. The possible realization of a complete software with graphical interface, that automatically handles the communications, through device protocols, only composing its model through a graphical editor program, is one of the most interesting achievements of this framework. This achievement has been realized using different methods, such as construction by configuration (C-b-C) and component based design (CBD). The centralized configuration management of the defined digital image of the process, which contains the communication data, permits the automatic generation of firmware dictionaries and software programs, provided with data maps that are synchronized with

the device ones. Defining these structures, using optimized editors, avoids the insertion of wrong instructions, during the development of these programs, for both firmware and software. It also drastically reduces development time. Compiling tables is a faster operation than writing variables exchange functions though APIs. The composition of GUIs using "drag and drop" features and proper selection lists, to link the GUI elements to the navigation structures, is a much faster operation rather than writing the code to compose windows, trees and behaviors, specific for the application. It could be considered a component based construction (CBC). The program structure allow to extend the available components, removing the design limits that are derived from the present selectable components. This task requires more additional coding, due to the need of coding also the editor part of the newly added element, but once it is done, its reuse is maximized and does not require any further coding activity, differently from the reuse though APIs. The software environment is extremely modular, as demonstrated by the possible usage of the Data Converters Layers to exploit only the protocol translation, which can be used as a separate API on which to build application logics. This has also been done recently to develop a supervisor prototype application for a particular machine, that has demonstrated the feasibility of the project and then to evaluate the development, with a proper supervisor unit like a PLC controller. The configuration management that has been described allows to remove and add variables to the process image, maintaining the compatibility between the various versions. Through the insertion of some identification data inside protocol maps the software can understand which version of the digital image is present inside the firmware program, and then use the proper configuration to interface with it. This aids the handling of firmware and software versioning, the configuration management and retro-compatibility. This feature is also very important for the prototyping of new devices, because in this stage a lot of versions are produced and a lot of elements are tested. The generation of a complete communication software in few steps gives extremely powerful debug and test capabilities. The interactions between software and firmware elements can be easily tested through consolidated elements. The realization of the mechanisms, that are the enabling elements of this approach, has been discussed in details in their design and construction. The same has

been done for the firmware part, that consists of libraries and modules, that are designed with proper patterns and strategies, to maximize the reuse of code, through an efficient abstraction from the hardware dependencies. The same concept of a modular CBD has been applied to these code modules. All the major firmware development aspects have been addressed with proper strategies. The abstraction mechanisms from the hardware, have brought to the introduction of the application drivers and modules layers, which decouple the hardware dependent development, from the application logic development. The application modules are built on data interfaces which are part of the digital image of the framework, that allow also their direct interaction with the software counterparts. It has been defined a generic firmware structure that is easily implementable though FSMs. Design patterns for modules, based on a common structure, that retrace device functioning, helps to maintain the coherence of the state of the program. The modular design grants reuse of code, but also a CBD approach in which these elements can be exchanged with similar ones, with few efforts. An innovative structure for the processing of control loops has been proposed (the XF-BAR), which can be also configured through its parametrization, allowing its reconfiguration also at run-time. This particular construction allows to achieve this reconfigurability, with mechanisms that can be exploited also in case of severe computational time constraints, such as motor control loops. The only drawback is a higher occupation of memory, due to the need of data structures, that contain the configuration of these mechanisms. All these firmware methods have been developed without the need of an object oriented design, which makes them very useful for bare-metal programs. They can be used also inside RTOSs, or others object oriented environments. These instruments have been developed mostly for devices which have FPUs, but a fixed-point implementation can be done using the same design. The D.A.E.D.A.L approach has been effectively used to reduce development time and also to develop interesting features, exploiting the interaction of the virtual components of embedded devices. The application on electric drives has led to very interesting results, that have also brought to the realization of commercial products. Electric drives, due to their characteristics and the need of complex commissioning softwares, are among the devices that can benefit most from this approach. The main purpose of this framework is to

reduce the efforts spent in repetitive tasks allowing the focusing on the design of new features. Its structure can be extended through the introduction of new features and components and the extension of supported protocols. New strategies can be developed such as the protocols synchronization. The potential of this approach has already been expressed, bringing some interesting results, but it is only the beginning of its possible evolution. As stated in the whole writing, embedded world is complex, and comprehends a lot of subjects and environments, that are in constant evolution, which need proper strategies. Hopefully the study of their design will continue to follow this evolution, trying to solve the emerging problems, with specific strategies. For what concerns future developments, there is an open universe of possibilities.

Appendix A

Cascade Control Improvements

As stated inside the whole writing, firmware development, comprehends many other disciplines that are needed to implement algorithms inside programs. In order to create a final and complete product all the necessary activities to develop programs and controls have to be done. This infrastructure aids the development of known algorithms and also the integration of new ones. In the realization of the various projects, hints for improvements were often encountered. Often most concepts are known in the theory but when implemented, sometimes practice differs from theory.

P.I.D discretization In the development of firmware for devices with FPU capabilities, micro-controllers manufacturers provide libraries for most of the known algorithms involved in motor control, optimized for their architectures. For examples, there are libraries with Clarke and Park direct and inverse transformations, P.I controllers and many digital filters. During the development of a firmware for motor control based on an ARM cortex M4F, that has a FPU unit, the provided P.I.D controllers were done without considering sampling time as parameter, that was simplified inside the tuning parameters of the P.I.D controller, in order to give a lighter implementation, from a computational point of view. The problem was that, for variable frequency drives, the sampling time of the execution of the P.I.D can change during its operation. That means that a different tuning should be done for every frequency

that has to be used. This solution is not very practical so in case of variable frequency drives a implementation that can parametrize the sampling time is preferable.

Implementation by ARM takes a different approach, using "normalized" integral and derivative gains: they are treated as independent from the sample time, but the author can manually change them. When control response equivalence with continuous time is searched for, integral gain supplied to ARM controller needs to be multiplied by sample time T s , while the derivative gain should be divided for the same quantity. [3]

Entering in the subject of algorithms discretization many options were encountered and evaluated, starting from discretization methods:

- Tustin.
- Backward-euler.
- Forward-euler.

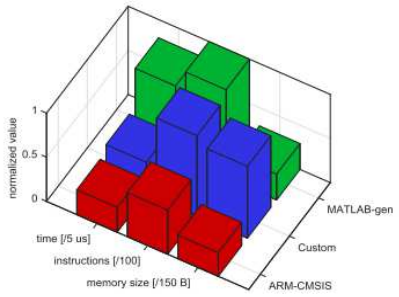


Figure A.1: Experimental evaluation of different figures of merit, by controller implementation. [3]

The result of this study was the development of a custom P.I.D library, which is the one developed inside the framework, written in C language, with the possibility of changing the sample time and many other features that are also the reason of

the higher memory requirements of this solution respect to the others, see fig. A.1. The "custom" implementation has a similar execution time as the ARM one, which is faster, also due to the usage of "normalized" gains. The important aspect that this work has shown is that when real implementations are compared there are many trade offs that affect this scenario and further analysis can be done. Another important result is the library designed with the proposed techniques, that has some useful features, such as the control response equivalence with continuous time.

Period and Frequency measurement In the development of motor control firmware, it also happen to encounter measurement problems. Another study that has been done is how to implement different measurement mechanisms exploiting hardware peripherals, and how sensor resolution impacts on the measures [4]. Inside this work were

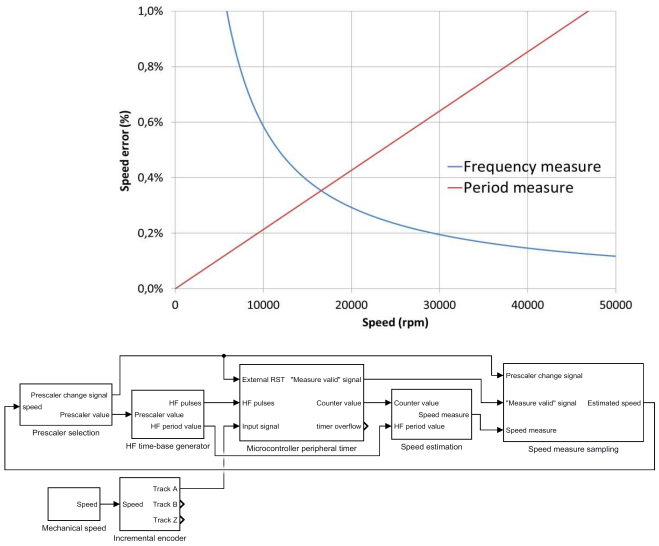


Figure A.2: Simulations and measurements comparison [4].

modeled, compared and implemented different speed measurement techniques to determine which mechanisms grant better performances with certain sensor resolutions and speed ranges. All these studies are realized inside the presented infrastructure in

which it is easy to develop these solutions and test them.

Auto-Phasing-Procedure Another important feature that has been developed inside the framework, is an automatic phasing procedure for hall and incremental encoder sensors. Hall sensor and incremental encoder with index reference are position sensors, that are used in the motor control to estimate rotor position, in order to apply the correct stator modulation, to produce motion. To have the maximum efficiency the modulation shall be done in a proper way, in which the magnetic fields generated by stator winding are always oriented to generate the maximum repulsion or attraction of the rotor magnetic field. To do that, the rotor orientation has to be determined, which can be done with these sensors. Hall sensors can be mounted inside the motor itself, on the phases windings, which determine an absolute and known reference of the rotor position. Often sensors are not built inside the motor, but they are mounted on the rotor shaft. The same thing is done for incremental encoders, with zero index reference. Incremental encoders are sensors which produce a certain number of pulses for every mechanical turn of the shaft, and the zero index is a reference that produces a pulse for every turn. Zero index and hall sensors are absolute references for the rotor position. These references are often aligned with a defined rotor orientation, during the mechanical mounting. There are many procedures to align these references to the predefined orientation, that align mechanically the sensor on the shaft. Often motors mounts incremental encoders with hall sensor emulation which are sensors that are able to give an absolute reference through the hall sensors, which have a low resolution, and an incremental information, with a higher resolution, through the encoder sensor. Absolute references are needed to properly estimate the rotor position to modulate the stator windings and they are needed when the device is powered up, and does not have any information about the rotor orientation. Without these references it has do to an alignment procedure at start up, which forces the rotor to a known position, see fig. A.3.

If these absolute references are not in the correct places, they cannot be used to estimate rotor position. That is the reason why they are mechanically aligned to a proper orientation. The developed procedure can automatically acquire these abso-

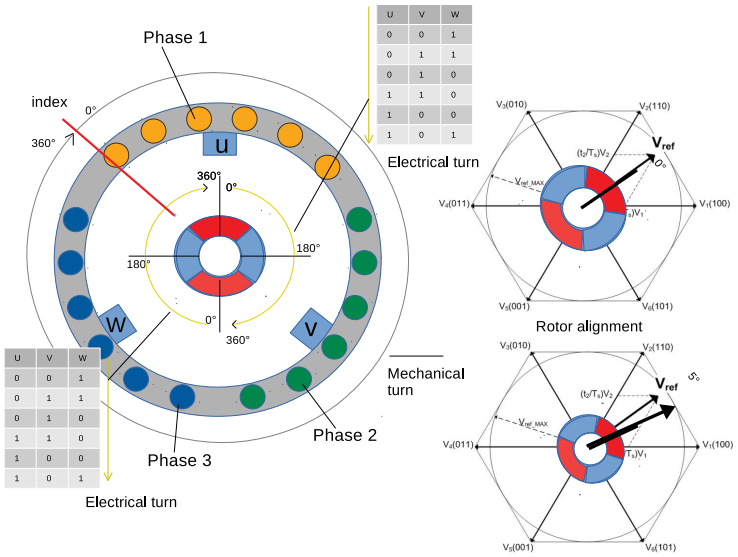


Figure A.3: Examples of electrical angle periodicity and absolute references for three phase brushless motor with a two pole pairs rotor.

lute references and their orientation respect to the rotor. The procedure works with all sensors, incremental encoders with hall sensor emulation, with or without zero index, and incremental encoder with zero index. The procedure consists of an automatic routine which forces the alignment condition of the rotor to an electrical angle corresponding to magnetic zero, using a percentage of the nominal current, that is enough to grant that the rotor can reach the alignment condition. Then the angle reference is incremented by small steps of a defined angular increment. Then the rotor is pulled to the moving reference and does a number of electrical turns that is equal to the pole pair number, in doing a complete mechanical turn. The procedure does two mechanical turns at most. During these turns when one of the absolute references is encountered, zero index crossing or hall transitions, the corresponding information is memorized, with the corresponding forced electrical angle information. In the case of hall sensor plus incremental encoder, the pulse count from the alignment condition is also memorized. That corresponds to the encoder pulses that occur between

the electrical zero of the magnetic field till that absolute reference. Memorizing the displacement of absolute references from the electrical zero and considering the pole pairs multiplicity it is possible to calculate position references with a very high precision. And also if there are some heterogeneous sensors, the procedure characterizes the specific mounted sensor. To use this procedure it is necessary to have a device with storage capabilities. The specific phasing routines are similar for these sensors, but they have a different handling. These stored references are used then in the motor control instead of predefined values. The implementation of these procedures is not treated in this work which is focused on the framework design, but this mechanism has been realized also exploiting framework mechanisms, especially configurable control loops and selectable sources and references through application driver layer.

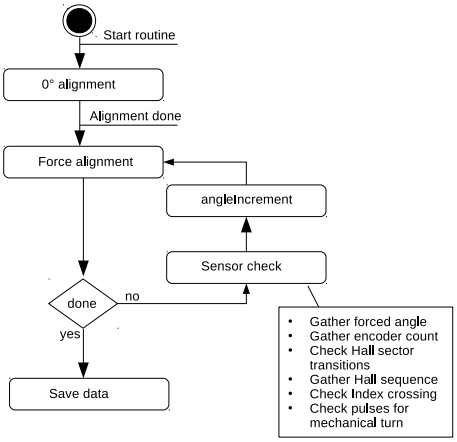


Figure A.4: Auto Phasing Algorithm.

Fig. A.4, shows the FSMs of the algorithm, which basically follows these steps:

- Generate a stator magnetic field to align the rotor with one of the electrical zeros.
- At periodic intervals, the electrical angle is incremented and a stator magnetic field, proper to align rotor to that angle, is generated. The time interval has

to be enough to align the rotor into the new position and the increment of electrical angle should be smaller than the sensors resolution. (For example in case of encoders with 2048 pulses per turn using quadrature signals, it should increment the angle no more than by $1/(2048)$ of one mechanical turn).

- During this procedure it is necessary to acquire: hall signal transitions from a sector to another, encoder pulse count, the forced electrical angle, hall sectors sequence, encoder direction count, index transition, which are operations that are already done in the motor control.

- These data are stored in a proper structure that according to the mounted sensor, during startup, configures the proper values that have to be used as references during the modulation of the magnetic field. For example, with hall sensor, when powered up, the electrical angle should be calculated as the mid of the two angle references stored during the transition between the two adjacent sectors. During hall transition angle reference should be calculated in the same way. For incremental sensors they can synchronize the angle with the corresponding stored value during index crossing. For encoder with hall sensor, both references can be used to synchronize the electrical angle, index stored reference and hall transitions.

This algorithm is a very important result and can be used to avoid mechanical alignment procedures in the sensor mounting, but using an automatic procedure that also can achieve a high precision. The only needed parameters are the sensor type, the motor pole pairs and the sensor resolution.

Appendix B

Industrial collaborations

As support to the experience matured in the industry, in this appendix are cited some of the most important collaborations that have been done with industries.

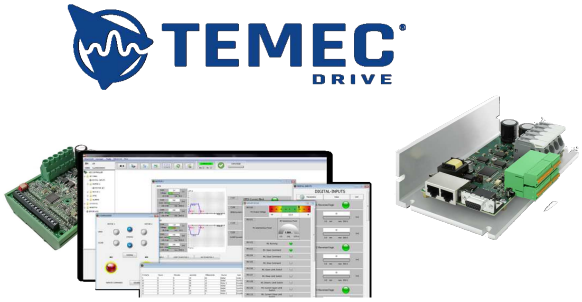


Figure B.1: Some of TeMec Drive products [5].

In fig.B.1, there is an overview of TeMec Drive products, that are commercial products, on which some of the presented techniques were adopted, both in the firmware and software design, during collaborations.
"TeMec Drive is a young and dynamic company focused on developing innovative and high technological products for the automation, especially for the control of elec-

trical motors. We offer wide-spectrum knowledge: every aspect, from hardware to PC software, is designed internally by our engineers, making us able to adapt to the customer needs and to create competitive and custom-made products.[5]"
This is the description of the TeMec company, taken from the "about" section in the website. Their engineering activities are mainly focused on the design and production of electric drives.

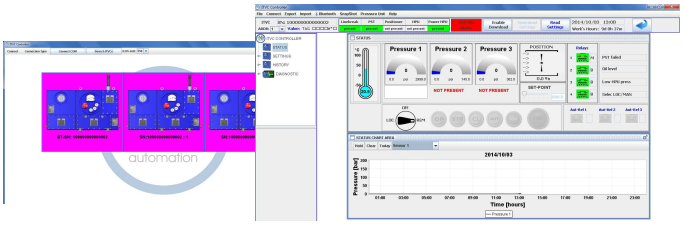


Figure B.2: Screen-shots of the program interface developed for DVG Automation.

In fig. B.2, there are some views of a software distributed by DVG Automation, that was developed during my first industrial collaboration. DVG Automation is a company that operates in different sectors, one of them is the flow control industry, especially for petrochemical industry. In this collaboration I have contributed to the development of the software interface program for a device applied to valve control systems. The collaboration with industries is a valuable presence inside the university and it has often been an opportunity to face on field problems and challenges, that contributed to the experience that has brought to the contents of this work.

B.1 EME case study

An important collaboration in which the proposed approach has permitted to achieve a significant results, was a project in collaboration with the EME Group, which is an Italian company established in 1983, leader in the production of electro-medical equipment for physiotherapy, aesthetics and aesthetic medicine. In this collaboration Eme, Tem an TeMecDrive, successfully cooperated in the realization of a prototype

of a physiotherapy leg press, of which the final design made by EME is shown in fig. B.3.

Figure B.3: EME leg press 983

In this collaboration a software customization has been developed in order to study the feasibility of the project and to realize the first functioning prototype of the application logic, fig. B.4. In the beginning of the work the software was intended to be used only for an initial stage of the development and then the plan was to exchange it with a PLC with an HMI panel. The first software application has been realized in few weeks, by a TeMec developer, that was trained in the use of this software framework. Without prior experience on communication protocols, he managed to realize a functional application, that implements the logic of the particular modes of this machine, exploiting the interaction with one of the electronic drives of TeMecDrive. This first prototype of application has proven to be reliable and due to the fast release time, has been chosen by EME for some further improvements and it was used as a demonstrator during one of their exhibitions. Even if the final product would not use this software as controller, the design approach has helped to document the process logic, abstracting it from its realization and the same logic will be used as reference in the final design that will be probably done using a PLC controller. In addition to that it has to be considered that the control logic inside the electric drive was realized through the design strategies proposed in section 4.2. And many of the operating modes of the machine have been developed also exploiting the interaction with the digital image of the control loop. This could be considered as a clear example of real application in which the use of both software and firmware methodologies has

brought an important contribution in the whole design of the system.

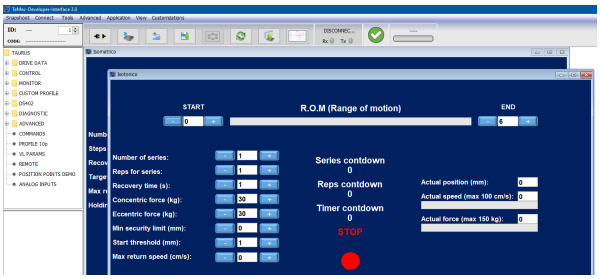


Figure B.4: EME software prototype done with S.I.E.G.Fr.I.E.D. APIs

B.2 Test applications case study

The framework is often used to realize automatic test procedures especially for the devices that are used with the generated commissioning software, but also for other products, as for example motor tests to check the correct assembling evaluating nominal characteristics within their range of tolerance.

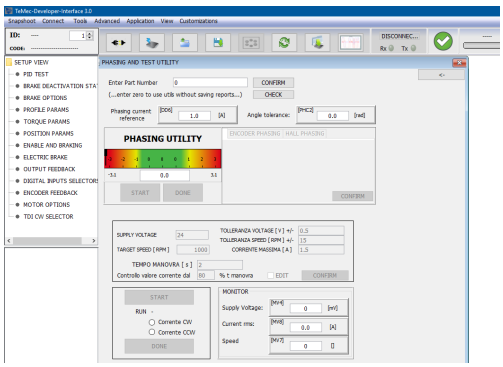


Figure B.5: Tem test procedure panel done with S.I.E.G.Fr.I.E.D. APIs

B.3 Commissioning software case study

The D.A.E.D.A.L approach shown in the previous sections, has demonstrated the capabilities of the firmware and software design. For what concern the software part, they have highlighted the possibility of using it as a middleware to build application upon it. This last case study shows a case in which the framework expresses its functionalities at best, that is the automatic generation of the commissioning software (which is synced with the generated firmware map). In this case all the communication part of the device and its relative software are prepared without writing any single line of code as it is for the first release of software for the AZ3 drive of TeMecDrive, see fig. B.6.

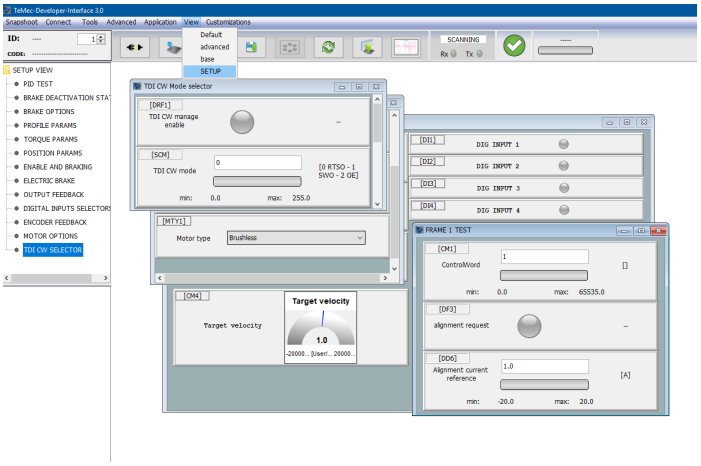


Figure B.6: Automatically generated and configured commissioning software through S.I.E.G.Fr.I.E.D.

Appendix C

License

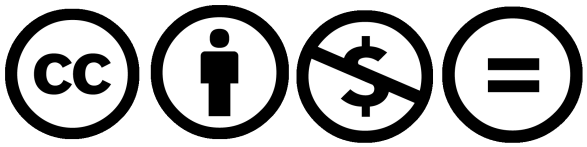


Figure C.1: CC BY-NC-ND 4.0

The author of this work is Zanichelli Roberto, which owns the rights on the content of this work and inherent materials.
My personal concerns about instruments that can realize complete products, is the tendency of using them to cut development costs. These tools are intended to aid development processes and not to be a substitute of software developers. These tools have to be used to let designers to focus on design tasks and not on constructions ones.

Eventual commercial concessions can be accorded, through a request to the au-

thor for consensus, under the granting of an ethical use of these tools. Without explicit consensus of the author this work must not be used for commercial purposes.

Bibliography

[1] Guido Matrella, Monica Mordonini, Roberto Zanichelli, and Riccardo Zini. The a.i.zeta framework: An ontological approach for aal systems control. In Filippo Cavallo, Vincenzo Marletta, Andrea Monteriù, and Pietro Siciliano, editors, *Ambient Assisted Living*, pages 223–238, Cham, 2017. Springer International Publishing.

[2] Sungjoo Yoo and Ahmed A. Jerraya. *Introduction to Hardware Abstraction Layers for SoC*, pages 179–186. Springer US, Boston, MA, 2003. URL: https://doi.org/10.1007/0-306-48709-8_14, doi:10.1007/0-306-48709-8_14.

[3] A. Soldati, R. Zanichelli, F. Brugnano, and C. Concari. Implementing discrete pid controllers: Benchmarking manual vs. automatic generation of embedded code. In *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society*, pages 178–183, Oct 2016. doi:10.1109/IECON.2016.7793334.

[4] F. Brugnano, C. Concari, E. Imamovic, F. Savi, A. Toscani, and R. Zanichelli. A simple and accurate algorithm for speed measurement in electric drives using incremental encoder. In *IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*, pages 8551–8556, Oct 2017. doi:10.1109/IECON.2017.8217502.

[5] Temec drive. URL: https://www.temecdrive.com/.

[6] CAN in Automation (CiA) e. V. *CiA ® 402 Draft Standard Proposal, Drives and motion control device profile Part 2: Operation modes and application data*. CiA, 2016. URL: https://www.can-cia.org/can-knowledge/canopen/cia402/.

[7] Michael Barr and Anthony Massa. *Programming Embedded Systems: With C and GNU Development Tools, 2nd Edition*. O'Reilly Media, 2006. URL: https://www.amazon.com/Programming-Embedded-Systems-Development-Tools/dp/0596009836?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0596009836.

[8] G. Berry. Challenges and potential solutions for complex embedded systems. In *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*, pages 1–1, Oct 2011. doi:10.1145/2038642.2038644.

[9] Mqx™ real-time operating system (rtos). URL: https://www.nxp.com/support/developer-resources/run-time-software/mqx-software-solutions/mqx-real-time-operating-system-rtos:MQXRTOS.

[10] L. D. Xu, W. He, and S. Li. Internet of things in industries: A survey. *IEEE Transactions on Industrial Informatics*, 10(4):2233–2243, Nov 2014. doi:10.1109/TII.2014.2300753.

[11] S. Charalampidou, A. Ampatzoglou, and P. Avgeriou. A process framework for embedded systems engineering. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 137–140, Aug 2014. doi:10.1109/SEAA.2014.58.

[12] Carliss Y. Baldwin and Kim B. Clark. *Modularity in the Design of Complex Engineering Systems*, pages 175–205. Springer Berlin Heidelberg, Berlin, Heidel-

berg, 2006. URL: https://doi.org/10.1007/3-540-32834-3_9, doi:10.1007/3-540-32834-3_9.

[13] I. Crnkovic. Component-based software engineering - new challenges in software development. In *Proceedings of the 25th International Conference on Information Technology Interfaces, 2003. ITI 2003.*, pages 9–18, June 2003. doi:10.1109/ITI.2003.1225314.

[14] Ivica Crnkovic, Brahim Hnich, Torsten Jonsson, and Zeynep Kiziltan. Specification, implementation, and deployment of components. *Commun. ACM*, 45(10):35–40, October 2002. URL: http://doi.acm.org/10.1145/570907.570928, doi:10.1145/570907.570928.

[15] C. Angelov and K. Sierszecki. A software framework for component-based embedded applications. In *11th Asia-Pacific Software Engineering Conference*, pages 655–662, Nov 2004. doi:10.1109/APSEC.2004.9.

[16] IEEE Computer Society. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. IEEE Computer Society Press, 2014. URL: https://www.amazon.com/Guide-Software-Engineering-Knowledge-SWEBOK/dp/0769551661?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0769551661.

[17] G. Jackelen and M. Jackelen. When standards and best practices are ignored. In *Proceedings 4th IEEE International Software Engineering Standards Symposium and Forum (ISESS'99). 'Best Software Practices for the Internet Age'*, pages 111–115, May 1999. doi:10.1109/SESS.1999.766584.

[18] About the unified modeling language specification version 2.5. URL: https://www.omg.org/spec/UML/2.5.

[19] J. Zhang, H. Rong, C. Zhang, J. Lu, F. Yan, and F. Xiang. A high dependability framework for real-time embedded control software design. In *2009 Interna-*

tional Symposium on Computer Network and Multimedia Technology, pages 1–4, Jan 2009. doi:10.1109/CNMT.2009.5374639.

[20] A. M. K. Cheng and S. S. Digewade. Design framework for self-stabilizing real-time systems based on real-time objects and prototype implementation with analysis. In *2009 International Conference on Embedded Software and Systems*, pages 499–504, May 2009. doi:10.1109/ICESS.2009.88.

[21] D. Selvyanti and Y. Bandung. The requirements engineering framework based on iso 29148:2011 and multi-view modeling framework. In *2017 International Conference on Information Technology Systems and Innovation (ICITSI)*, pages 128–133, Oct 2017. doi:10.1109/ICITSI.2017.8267930.

[22] P. Karanth. Design of a framework for knowledge analytics. In *2016 22nd Annual International Conference on Advanced Computing and Communication (ADCOM)*, pages 44–47, Sept 2016. doi:10.1109/ADCOM.2016.17.

[23] Ralph E. Johnson. Frameworks = (components + patterns). *Commun. ACM*, 40(10):39–42, October 1997. URL: http://doi.acm.org/10.1145/262793.262799, doi:10.1145/262793.262799.

[24] Wolfgang Pree, Gustav Pomberger, and Franz Kapsner. Framework component systems: Concepts, design heuristics, and perspectives. In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Perspectives of System Informatics*, pages 330–340, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[25] Suhardi, P. M. Budhiputra, and P. Yustianto. Service engineering framework: A simple approach. In *2014 International Conference on Information Technology Systems and Innovation (ICITSI)*, pages 130–134, Nov 2014. doi:10.1109/ICITSI.2014.7048251.

[26] V. Okanović. Designing a web application framework. In *2011 18th International Conference on Systems, Signals and Image Processing*, pages 1–4, June 2011.

[27] Y. Saito, H. Aoki, J. Sato, and D. C. V. Khuong. Software framework for embedded digital consumer appliance. In *2011 IEEE 15th International Symposium on Consumer Electronics (ISCE)*, pages 528–531, June 2011. doi:10.1109/ISCE.2011.5973886.

[28] A. Hamou-Lhadj and A. Hamou-Lhadj. An organizational framework for building secure software. In *2008 International Conference on Information Security and Assurance (isa 2008)*, pages 457–460, April 2008. doi:10.1109/ISA.2008.105.

[29] C. Angelov, K. Sierszecki, and F. Zhou. A software framework for hard real-time distributed embedded systems. In *2008 34th Euromicro Conference Software Engineering and Advanced Applications*, pages 385–392, Sept 2008. doi:10.1109/SEAA.2008.29.

[30] Pao-Ann Hsiung, Feng-Shi Su, Chuen-Hau Gao, Shu-Yu Cheng, and Yu-Ming Chang. Verifiable embedded real-time application framework. In *Proceedings Seventh IEEE Real-Time Technology and Applications Symposium*, pages 109–110, May 2001. doi:10.1109/RTTAS.2001.936258.

[31] M. H. Khan, Y. Bai, B. Xiao, and P. N. Vaidya. A highly accurate power and performance scaling framework in an embedded environment. In *2007 Digest of Technical Papers International Conference on Consumer Electronics*, pages 1–2, Jan 2007. doi:10.1109/ICCE.2007.341560.

[32] N. Vun, K. Xu, and Z. Foo. A modular framework for applications development on embedded platforms. In *2007 IEEE International Symposium on Consumer Electronics*, pages 1–6, June 2007. doi:10.1109/ISCE.2007.4382144.

[33] T. Seceleanu and G. Sapienza. A tool integration framework for sustainable embedded systems development. *Computer*, 46(11):68–71, Nov 2013. doi:10.1109/MC.2013.297.

[34] J. Zhang, F. Yan, C. Zhang, H. Rong, and F. Xiang. High dependability design framework in real-time embedded control software. In *2009 Second Interna-*

tional Symposium on Computational Intelligence and Design, volume 2, pages 547–551, Dec 2009. doi:10.1109/ISCID.2009.283.

[35] H. Lei and Y. Wang. A model-driven testing framework based on requirement for embedded software. In *2016 11th International Conference on Reliability, Maintainability and Safety (ICRMS)*, pages 1–6, Oct 2016. doi:10.1109/ICRMS.2016.8050100.

[36] P. Iyenghar, E. Pulvermueller, C. Westerkamp, and J. Wuebbelmann. Integrated model-based approach and test framework for embedded systems. In *FDL 2011 Proceedings*, pages 1–8, Sept 2011.

[37] P. Iyenghar. Test framework generation for model-based testing in embedded systems. In *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 267–274, Aug 2011. doi:10.1109/SEAA.2011.48.

[38] Jing Zhang, Fenghong Xiang, Bin Wang, and Jing Lu. An extensible software framework for reliable distributed embedded system modeling. In *2010 2nd International Asia Conference on Informatics in Control, Automation and Robotics (CAR 2010)*, volume 2, pages 234–237, March 2010. doi:10.1109/CAR.2010.5456558.

[39] J. K. R. Sastry, V. C. Prakash, and L. S. S. Reddy. A formal framework for verification and validation of external behavioral models of embedded systems through use case models. In *2009 Fourth International Conference on Embedded and Multimedia Computing*, pages 1–8, Dec 2009. doi:10.1109/EM-COM.2009.5402964.

[40] P. . Hsiung, T. . Lee, J. . Fu, and W. . See. Formal verification of real-time embedded software in an object-oriented application framework. *IEE Proceedings - Computers and Digital Techniques*, 151(6):417–434, Nov 2004. doi:10.1049/ip-cdt:20041102.

[41] Y. Yin, B. Liu, and G. Zhang. On framework oriented embedded software testing development environment. In *2009 8th International Conference on Reliability, Maintainability and Safety*, pages 708–712, July 2009. doi:10.1109/ICRMS.2009.5270096.

[42] F. E. Peter and K. G. Blemel. A low cost embedded instrumentation (ei) framework for vehicle health management systems (vhms). In *2008 IEEE Aerospace Conference*, pages 1–5, March 2008. doi:10.1109/AERO.2008.4526638.

[43] P. Putthapipat and K. Techakittiroj. Piframe: A framework for home automation platform on the full feature os. In *2016 International Conference on Electronics, Information, and Communications (ICEIC)*, pages 1–4, Jan 2016. doi:10.1109/ELINFOCOM.2016.7562978.

[44] *Development Guidelines for Vehicle Based Software*. Motor Industry Research Association, 1994. URL: https://www.amazon.com/Development-Guidelines-Vehicle-Based-Software/dp/0952415607?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0952415607.

[45] Honeywell, variable frequency drive pc tool software. URL: https://customer.honeywell.com/en-US/support/commercial/software/vfds/Pages/default.aspx.

[46] Abb, software tools. URL: https://new.abb.com/drives/software-tools.

[47] Mcuxpresso software and tools for arm® cortex®-m cores. URL: https://www.nxp.com/support/developer-resources/software-development-tools/mcuxpresso-software-and-tools:MCUXPRESSO.

[48] merriam-webster. URL: https://www.merriam-webster.com/dictionary/daedal.

[49] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994. URL: http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1.

[50] ISO/IEC JTC 1/SC 7 Software and systems engineering. Iso/iec/ieee 24765:2017 systems and software engineering – vocabulary, 2017.

[51] E. C. Groen, N. Seyff, R. Ali, F. Dalpiaz, J. Doerr, E. Guzman, M. Hosseini, J. Marco, M. Oriol, A. Perini, and M. Stade. The crowd in requirements engineering: The landscape and challenges. *IEEE Software*, 34(2):44–52, Mar 2017. doi:10.1109/MS.2017.33.

[52] Agile manifesto. URL: http://agilemanifesto.org.

[53] S. J. Khalaf and K. A. Maria. An empirical study of xp: The case of jordan. In *2009 International Conference on Information and Multimedia Technology*, pages 380–383, Dec 2009. doi:10.1109/ICIMT.2009.94.

[54] S. Thakur and H. Singh. Fdrd: Feature driven reuse development process model. In *2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies*, pages 1593–1598, May 2014. doi:10.1109/ICACCCT.2014.7019376.

[55] K. Rapley. Rad or trad or both? the future of software development. In *IEE Colloquium on Will Tickit and ISO 9000 Survive Rapid Application Development?*, pages 3/1–3/2, Dec 1995. doi:10.1049/ic:19951554.

[56] J. Coplien. Patterns of engineering. *IEEE Potentials*, 23(2):4–8, April 2004. doi:10.1109/MP.2004.1289991.

[57] I. Sommerville. Software construction by configuration: challenges for software engineering research. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, page 9. IEEE, 2005. URL: http://ieeexplore.ieee.org/document/1510097/, doi:10.1109/ICSM.2005.81.

[58] C. Zhang and W. Shen. The studies on integrated development approach based on the architecture and mvc design pattern. In *2012 IEEE Symposium on Electrical Electronics Engineering (EEESYM)*, pages 578–580, June 2012. doi:10.1109/EEESym.2012.6258723.

[59] Guangquan Zhang and Mei Rong. Exploration and practice of software engineering core curriculum construction. In *2011 6th International Conference on Computer Science & Education (ICCSE)*, pages 703–705. IEEE, aug 2011. URL: http://ieeexplore.ieee.org/document/6028734/, doi:10.1109/ICCSE.2011.6028734.

[60] S. McConnell. Closing the gap. *IEEE Software*, 19(1):3–5, jan 2002. URL: http://ieeexplore.ieee.org/document/976933/, doi:10.1109/MS.2002.976933.

[61] H. Zhu and I. Bayley. On the composability of design patterns. *IEEE Transactions on Software Engineering*, 41(11):1138–1152, Nov 2015. doi:10.1109/TSE.2015.2445341.

[62] S. McConnell. Who cares about software construction? *IEEE Software*, 13(1):127–128, 1996. URL: http://ieeexplore.ieee.org/document/476305/, doi:10.1109/52.476305.

[63] W. Suryn, F. Robert, A. Abran, P. Bourque, and R. Champagne. Experimental support analysis of the software construction knowledge area in the swebok guide. In *10th International Workshop on Software Technology and Engineering Practice*, pages 193–197, Oct 2002. doi:10.1109/STEP.2002.1267632.

[64] Souheil Khaddaj, Yajna Kumar Makoondlall, Bippin Makoond, Shyam Chivukula, and Kethan Keerthi. ZDLC: Towards Automated Software Construction and Migration. In *2015 14th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)*, pages 13–16. IEEE, aug 2015. URL: http://ieeexplore.ieee.org/document/7429545/, doi:10.1109/DCABES.2015.11.

[65] G. K. Palshikar. Applying formal specifications to real-world software development. *IEEE Software*, 18(6):89–97, Nov 2001. doi:10.1109/52.965810.

[66] Roel Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Comput. Surv.*, 30(4):459–527, December 1998. URL: http://doi.acm.org/10.1145/299917.299919, doi:10.1145/299917.299919.

[67] W. G. Howerton and M. G. Hinchey. Using the right tool for the job. In *Proceedings Sixth IEEE International Conference on Engineering of Complex Computer Systems. ICECCS 2000*, pages 105–115, Sept 2000. doi:10.1109/ICECCS.2000.873932.

[68] Eclipse foundation, the platform for open innovation and collaboration. URL: https://www.eclipse.org/.

[69] Eclipse (software) from wikipedia, the free encyclopedia. URL: https://en.wikipedia.org/wiki/Eclipse_(software).

[70] Sw4stm32. URL: https://www.st.com/en/development-tools/sw4stm32.html.

[71] M. V. Cherkasskiy and A. O. Melnyk. Modern content of algorithm theory. In *Modern Problems of Radio Engineering, Telecommunications and Computer Science (IEEE Cat. No.02EX542)*, pages 274–, Feb 2002. doi:10.1109/TCSET.2002.1015961.

[72] N. Jain, S. G. Mali, and S. Kulkarni. Infield firmware update: Challenges and solutions. In *2016 International Conference on Communication and Signal Processing (ICCSP)*, pages 1232–1236, April 2016. doi:10.1109/ICCSP.2016.7754349.

[73] A. Varghese and A. K. Bose. Threat modelling of industrial controllers: A firmware security perspective. In *2014 International Conference on Anti-Counterfeiting, Security and Identification (ASID)*, pages 1–4, Dec 2014. doi:10.1109/ICASID.2014.7064951.

[74] B. Peischl. On business drivers for firmware test: A wake-up call for software engineering research? In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 593–594, July 2017. doi:10.1109/QRS-C.2017.109.

[75] Interactive: The top programming languages 2018, ieee spectrum 2018. URL: https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018.

[76] Ling Ming, Shi Longxing, and Tang Yongming. An advanced c programming course for the embedded systems development. In *Proceedings of IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE) 2012*, pages H2A–1–H2A–3, Aug 2012. doi:10.1109/TALE.2012.6360328.

[77] C. Ebert and C. Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, April 2009. doi:10.1109/MC.2009.118.

[78] S. Nuratch. A universal microcontroller circuit and firmware design and implementation for iot-based realtime measurement and control applications. In *2017 International Electrical Engineering Congress (iEECON)*, pages 1–4, March 2017. doi:10.1109/IEECON.2017.8075906.

[79] A. I. Laperdin and V. D. Yurkevich. Development of an universal firmware for control of testbed in strength laboratory. In *2016 13th International Scientific-*

Technical Conference on Actual Problems of Electronics Instrument Engineering (APEIE), volume 03, pages 1–1, Oct 2016. doi:10.1109/APEIE.2016.7806992.

[80] Bruce Powel Douglass. *Design Patterns for Embedded Systems in C: An Embedded Software Engineering Toolkit*. Newnes, 2010. URL: https://www.amazon.com/Design-Patterns-Embedded-Systems-Engineering-ebook/dp/B004FGMTTK?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=B004FGMTTK.

[81] Michael J. Pont. *Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers (with CD-ROM)*. Addison-Wesley, 2001. URL: https://www.amazon.com/Patterns-Time-Triggered-Embedded-Systems-Microcontrollers/dp/0201331381?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0201331381.

[82] Deniz Akdur, Onur Demirors, and Vahid Garousi. Characterizing the development and usage of diagrams in embedded software systems. In *Proceedings - 43rd Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2017*, pages 167–175. IEEE, aug 2017. URL: http://ieeexplore.ieee.org/document/8051344/, doi:10.1109/SEAA.2017.13.

[83] O. Rafique, M. Gesell, and K. Schneider. Targeting different abstraction layers by model-based design methods for embedded systems: A case study. In *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 334–337, Aug 2013. doi:10.1109/RTCSA.2013.6732235.

[84] G. Krivulya, E. Kovalyov, O. Korobko, and M. Laptev. The analysis of modern fsm description languages. In *2006 International Conference - Modern Problems of Radio Engineering, Telecommunications, and Computer Science*, pages 370–374, Feb 2006. doi:10.1109/TCSET.2006.4404555.

[85] D. Hooshyar, R. B. Ahmad, M. H. N. M. Nasir, and W. C. Mun. Flowchart-based approach to aid novice programmers: A novel framework. In *2014 International Conference on Computer and Information Sciences (ICCOINS)*, pages 1–5, June 2014. doi:10.1109/ICCOINS.2014.6868826.

[86] S. Xinogalos. Using flowchart-based programming environments for simplifying programming and software engineering processes. In *2013 IEEE Global Engineering Education Conference (EDUCON)*, pages 1313–1322, March 2013. doi:10.1109/EduCon.2013.6530276.

[87] Thomas A. Henzinger and Christoph M. Kirsch. The embedded machine: Predictable, portable real-time code. *ACM Trans. Program. Lang. Syst.*, 29(6), October 2007. URL: http://doi.acm.org/10.1145/1286821.1286824, doi:10.1145/1286821.1286824.

[88] Cia® 402 series: Canopen device profile for drives and motion control. URL: https://www.can-cia.org/can-knowledge/canopen/cia402/.

[89] R. Martinho, J. Varajao, and D. Domingos. A two-step approach for modelling flexibility in software processes. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 427–430, Sep. 2008. doi:10.1109/ASE.2008.65.

[90] C. Ackermann, M. Lindvall, and G. Dennis. Redesign for flexibility and maintainability: A case study. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 259–262, March 2009. doi:10.1109/CSMR.2009.60.

[91] S. Peng, L. Shen, H. Liu, and F. Li. User-oriented measurement of software flexibility. In *2009 WRI World Congress on Computer Science and Information Engineering*, volume 7, pages 629–633, March 2009. doi:10.1109/CSIE.2009.207.

[92] L. Zhang, L. Li, and H. Gao. 2-d software quality model and case study in software flexibility research. In *2008 International Conference on Computational Intelligence for Modelling Control Automation*, pages 1147–1152, Dec 2008. doi:10.1109/CIMCA.2008.70.

[93] P. G. Xing, T. Qi, W. Ya, and Y. Yi. "flexibility" of software development method. In *2013 Fourth International Conference on Digital Manufacturing Automation*, pages 620–622, June 2013. doi:10.1109/ICDMA.2013.146.

Ringraziamenti

In primo luogo ringrazio il mio tutor, che mi ha dato la possibilità di intraprendere questo percorso molto particolare e ha sempre creduto che questa particolarità andasse incentivata. Ringrazio inoltre gli amici e colleghi del laboratorio di automazione industriale e i docenti che hanno creduto in me e mi hanno dato la possibilità di imparare. Ringrazio studenti e compagni che hanno accolto il mio modo di pensare e che hanno contribuito ad arricchire la mia esperienza. Ringrazio le varie persone con cui ho collaborato in questi anni, tecnici, ricercatori e progettisti nei vari ambiti di ricerca e industria. Ultimi, ma non per importanza, ringrazio tutti gli amici, vecchi e nuovi, che fanno parte della mia vita, chi mi conosce da sempre e chi da poco, ma che contribuiscono a rendere speciali i momenti passati assieme. Ringrazio gli amici che condividono le mie passioni, dallo sport ai giochi di società. Infine rigrazio la mia famiglia, in particolar modo mia madre, che è la persona che più di tutti è stata sempre in prima linea nel sostenere il mio percorso che mi ha portato a diventare la persona che sono oggi.