Janne Partanen

Application for controlling PICkit multiprogrammers

Bachelor's thesis

Bachelor of Engineering

Information Technology/Game Programming

June 2020

Kaakkois-Suomen ammattikorkeakoulu

Tekijä/Tekijät	Tutkintonimike	Aika
Janne Partanen	Insinööri (AMK)	Kesäkuu 2020
Opinnäytetyön nimi Ohjelma PICkit-moniohjelmoijien hallintaan		46 sivua 15 liitesivua
Toimeksiantaja Produal Oy		
Ohjaaja Teemu Saarelainen		

Tiivistelmä

Tämän opinnäytetyön tavoitteena oli kehittää ja ottaa käyttöön ensimmäinen toimiva versio ohjelmasta, joka ohjaa PICkit-moniohjelmoijia käyttämällä hyväkseen Microchipin kehittämää komentorivi-käyttölittymää. Ohjelmaan luotiin lisäksi graafinen käyttöliittymä ja ohjelma ohjelmoitin Pythonilla. Opinnäytetyön toimeksiantajana toimi Produal Oy ja ohjelman kehitys tapahtui joulukuusta 2019 maaliskuuhun 2020. Produal Oy on yritys, jonka erikoisuutena on rakennuksissa olevien muuttujien, kuten hiilidioksidia tai lämpötilaa mittaavien tuotteiden luonti. Nämä tuotteet ohjaavat rakennusautomaation niihin liittyviä osia.

Opinnäytetyön tarkoituksena on tarjota tietoa ohjelman kehitysprosessista ja ominaisuuksista. Ohjelma tarjoaa Produalin tuotantotyöntekijöille tavan hallita yrityksen moniohjelmoijia mahdollisimman vähällä käyttäjäsyötteellä ja antaa moniohjelmoijille joustavuutta, joka sallii niiden käytön kaikille Produalin tuotteille. Opinnäytetyö kuvaa myös yleiset ohjelmistotuotannon periaatteet, joita käytetään ohjelmistoa luotaessa, ja kuinka niitä käytetiin opinnäytetyöohjelman luonnissa.

Opinnäytetyö saatiin valmiiksi Produalin vaatimusten mukaisesti. Kaikki Produalin kanssa sovitut ominaisuudet toteutettiin ohjelmaan ja Produalin tuotantotiimi on ottannut ohjelman käyttöönsä.

Opinnäytetyön litteet ovat salassapidettäviä.

Asiasanat

Python, PICkit, ohjelma, graafinen käyttöliittymä, työpöytä

Author (authors)	Degree	Time
Janne Partanen	Bachelor of Engineering	June 2020

Thesis title

Application for controlling PICkit multiprogrammers

46 pages
15 pages of appendices

Commissioned by

Produal Ltd.

Supervisor

Teemu Saarelainen

Abstract

The objective of this thesis was to develop and deploy the first operational version of an application that can control PICkit multiprogrammers by using Microchip's own command line interface. A graphical user interface was created for the application and the programming was done with Python. The thesis was commissioned by Produal Ltd, and the development process of the application took place from December 2019 to March 2020. Produal Ltd is a company that specializes in the creation of products that measure variables in buildings, such as CO2 and temperature and control the related parts of building automation.

This thesis provides insight into the development process and features of the application. The application provides Produal's production employees with a way to control the company's multiprogrammers with minimal user input and flexibility that allows the multiprogrammers to be used for all of Produal's products. This thesis also describes, in general, software engineering practices used when creating an application and more specifically the practices used in the creation of the commissioner's application.

The application was finished according to the specifications that were made in cooperation with Produal. All the features that were agreed upon with Produal were implemented into the application, and the application has been taken into use by Produal's production team.

The appendices of the thesis are confidential.

Keywords

Python, PICkit, application, GUI

Table of Contents

APPENDICES

TERMS AND ABBREVIATIONS

ATM	Automatic Teller Machine. A machine that can be used to retrieve physical money.
CNL	Current Number in Loop. Refers to the current number in the loop that creates subprocesses from 1 to the chosen number of PICkits.
FURPS+	An abbreviation that means: Functionality, Usability, Reliability, Performance, Supportability. Refers to non-functional system properties. Marsic (2012, 75.)
GUI	Graphical User Interface. An interface that directs the software through an interactive interface where selections are made using a mouse (Bell 2005, 54-55)
HEX files	Files that contain data in a hexadecimal format typically used by programmable logic devices, such as microcontrollers.
IDE	Integrated Development Environment. Programming environments that combine different parts to one suite, like building executable files and editing the source code
IEEE	Institute of Electrical and Electronics Engineers
IPE	Refers to MPLAB X IPE
IPECMD	Command line interface included with the MPLAB X IPE created by Microchip Technology Inc. Provides the same functionalities as the MPLAB X IPE but is used through the command line.
microSDHC	micro Secure Digital High Capacity. A type of memory card based on Flash memory that offers higher data capacity (4-16 GB) compared to normal SD cards (64 MB-2 GB) (GSMArena no date a; GSMArena no date b).

MPLAB X IDE	Integrated Development Environment created by Microchip Technology Inc. Used to develop applications for Microchip's microcontrollers (Microchip 2019).
MPLAB X IPE	Integrated Programming Environment created by Microchip Technology Inc. Allows for the simple use of programming features (Microchip 2020a).
PICkit	PICkit is a series of physical in-circuit debugger and programmer products produced by Microchip Technology Inc for the purpose of programming microcontrollers (Microchip 2020b).
PICkit multiprogrammer	Multiple PICkits connected to a computer through a USB hub
SDD	Software Design Description/Document. The SDD provides a model of the software that is being developed (IEEE-SA Standards Board).
SRS	Software Requirements Specification document. A document that is meant to provide developers a detailed description about the system that is to be implemented (Sommerville 2016, 126-127)
SVN	Subversion. A version control system that is used to manage and track changes to assets in a project.
USB	Universal Serial Bus. A type of computer port used to connect devices, such as printers and keyboards to a computer (TechTerms no date).

1 INTRODUCTION

The objective of this thesis was to develop a program that allows controlling PICkit multiprogrammers directly from the user's computer. The thesis was commissioned by Produal Ltd, a company that specializes in the creation of products that measure variables in buildings, such as CO2, temperature etc. and control the related parts of building automation (About us). The company has offices in seven countries and employs approximately 100 people (Basic Facts).

The development cycle of the application took place from December 2019 to the end of March 2020. The commissioner’s production staff wanted to have a way to reprogram the PICkits in their multiprogrammers (which are multiple PICkits connected to a computer through an USB hub) and be able to program their products directly. PICkits are physical in-circuit debugger and programmer products produced by Microchip Technology Inc for the purpose of programming microcontrollers (Microchip 2020b). The problem the production staff faced was that should they want to change the software contained inside PICkits, they needed to take the multiprogrammers apart and reprogram the PICkits one by one manually by using MPLAB X IPE. Therefore, they wanted an application that could accomplish the aforementioned tasks with minimal effort from the production employees.

This thesis aims to illustrate the development process and features of the application using the Python programming language in conjunction with Microchip’s own command line interface for PICkit debuggers/programmers and a graphical user interface library as well as the general practices of software development. The application provides a way to program and direct the operation of PICkit multiprogrammers through MPLAB’s IPECMD command line interface.

The application was created with the intention of having it used by Produal’s production employees. The program also aimed to lessen the time used in the reprogramming of the PICkit debuggers by allowing them to be programmed easily with minimal input from the user. The application also allows the production employees to focus on other tasks while the application is running as

the graphical feedback the application provides will tell them when the programming is complete, and the application can also be used to directly program Produal's products.

The first chapter focuses on the basic software engineering practices and how they were followed in the creation of the commissioner's application. The structure of the chapter's contents are primarily based on Sommerville's (2016) book. The second chapter covers the development tools and technologies that were used in the development of the application. The third chapter describes the implementation of the application itself. The fourth chapter is focused on the testing and deployment process of the application. The fifth chapter introduces features that could be possibly added to updated versions of the application. The final chapter discusses the author's personal reflection on the project, focusing on what could have been done better, what was done well, what goals were achieved for the application and what the author learned during the development process.

Some of the pictures in this thesis are only partially presented and some details of the implementation process are intentionally. This is due to a confidentiality and non-disclosure agreement made between the author and Produal Ltd to prevent revealing any sensitive information about the company's operating methods.

2 SOFTWARE ENGINEERING PRACTICES

This chapter describes the general steps in software engineering that are used when developing an application and the use of these principles in the making of the commissioner's application. The following sub-chapters generally describe the steps in the waterfall model of software development, where one step usually produces a specified output such as documents before moving on to the next step in the model (Sommerville 2016, 48). Though the commissioner's application development did not follow all the principles of the waterfall model, it is an appropriate reference since, after all the software interacted with hardware which, according to Sommerville (2016, 49), justifies the use of the waterfall model. The waterfall model was also chosen for the commissioner's application because it presents a logical development path in

which each step produces something that can be used in the next step and, in general, following the model helps ensure that the client receives the product they desire.

The most common steps in the waterfall model are:

1. Requirements specification
2. Design
3. Implementation
4. Testing
5. Deployment
6. Maintenance

There can be more steps, and each of these steps involve specific processes that produce outputs. One step that often takes place before requirements specification involves defining how the process of developing an idea for a software application could be approached. Also, it should be determined if it is possible or worth the effort to start processing an idea further for example by comparing the costs to the benefits provided by the software. This step took place during the development cycle of the commissioner's application. There is no single correct waterfall model. For example, Sommerville defines the steps in the waterfall model as follows (Figure 1).

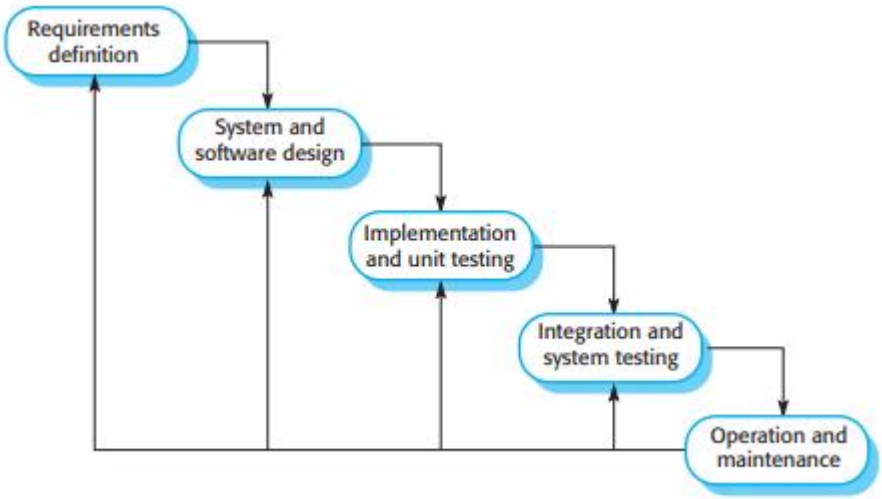

Figure 1: Waterfall model (Sommerville 2016, 47).

2.1 Requirements specification

The software developmental cycle generally begins with the process of determining what the user wants the software to do. This process is called requirements specification.

As Sommerville (2016, 102-103) states, there are two types of requirements involved in the requirements specification process. The first type is user requirements. These requirements are usually statements that aim to tell what the software should do and what kind of constraints it should have. Depending on the user making the requirement statement, they can vary from broad descriptions to extremely detailed specifications,since very often the more technically minded writer, the more detailed the requirement statement. The second type of requirements are system requirements. These types of requirements describe in detail the flow of defining how a feature set in the user requirement should function in relation to the rest of the system.

System requirements are further divided into two subcategories: functional and non-functional requirements. Functional requirements specify aspects related to the program's functions, such as the features it should have, how it should behave and how it should respond to different inputs. They can also state what the software should not do. In short, they determine what the software should and should not do. (Sommerville 2016, 105.) Non-functional requirements, on the other hand, tend to specify the attributes of the software that help define how the product works. For example, the operating speed and capacity are non-functional requirements. (Functional Requirements vs Non-Functional Requirements: Key Differences no date.) Marsic (2012, 75) describes the concept of **FURPS+** with regards to the non-functional properties of a system in the following way:

- *Functionality lists additional functional requirements that might be considered, such as security, which refers to ensuring data integrity and authorized access to information*
- *Usability refers to the ease of use, esthetics, consistency, and documentation—a system that is difficult and confusing to use will likely fail to accomplish its intended purpose*
- *Reliability specifies the expected frequency of system failure under certain operating conditions, as well as recoverability, predictability, accuracy, and mean time to failure*

- *Performance details the computing speed, efficiency, resource consumption, throughput, and response time*
- *Supportability characterizes testability, adaptability, maintainability, compatibility, configurability, installability, scalability, and localizability*

Also, all requirements must be testable, meaning they must be specified in a way that makes writing tests for them easy since if tests cannot be written for the different requirements, the requirements become difficult if not impossible to implement (Marsic 2012, 76).

Bell (2005, 38) provides a description of the components of a good requirement specification:

- Implementation free - the specification only describes what is needed
- Complete - nothing is missing from the specification
- Consistent - the requirements do not contradict each other
- Unambiguous - the specification cannot be misinterpreted
- Concise - no requirement is repeated
- Minimal - no unnecessary elements in the specification
- Understandable - both developer and user can understand the requirement specification
- Achievable – the requirement can be implemented
- Testable – tests can be made for the requirement

Eliciting the requirements refers to the process of gathering the user and system requirements through different methods. In part, this stage is complete when plans are made for creating software as some of the functional requirements are already outlined before the software development process even begins. There are three different ways that specifications are created:

1. Requirement elicitation, where the developer and the user/client converse about the requirements and clarify them, with the result creating a specification (Bell 2005, 40).
2. Requirement analysis, where the developer/analyst refines the requirements gathered from the client. Analysis also involves the creation of user scenarios that tell how the user interacts with different parts of the system (Marsic 2012, 69).
3. Requirement definition, where only the requirement specification must be written.

In general, there are two different ways a requirement specification might be written. The first is specification in natural language, which involves writing out plainly in the writer's own language what they want the specified feature to do. However, this method of writing the specifications has the risk of being very vague about what is needed. In order to mitigate this, it is recommended that the specification written in a natural language is easy to read for someone

who might not understand technical language, differentiates mandatory and optional requirements with “shall” and “should” respectively and includes the reason for the requirement. (Sommerville 2016, 121-122.)

The second way a specification might be written is a structured approach. The structured approach to writing requirements helps remove some of the vagueness that simply writing out the specification can bring.The usage of this method requires the creation of templates to provide the structure of the specification. (Sommerville 2016, 123.) In his book, Sommerville (2016, 123) states that “the specification may be structured around the objects manipulated by the system, the functions performed by the system, or the events processed by the system.” This approach to writing specifications makes it easier for multiple people to write requirements as they follow the same structure and for the developer to understand the requirement. However, the structured approach does not work as well for requirements that have their full scope defined during the implementation process.

Use cases define how different types of users interact with the program using a diagram to show the different users or user types and functionalities in the system. It is recommended that each use case is documented with a detailed written description about what the logical process of the use case is (Sommerville 2016, 125). One example of a use case diagram is shown in Figure 2.

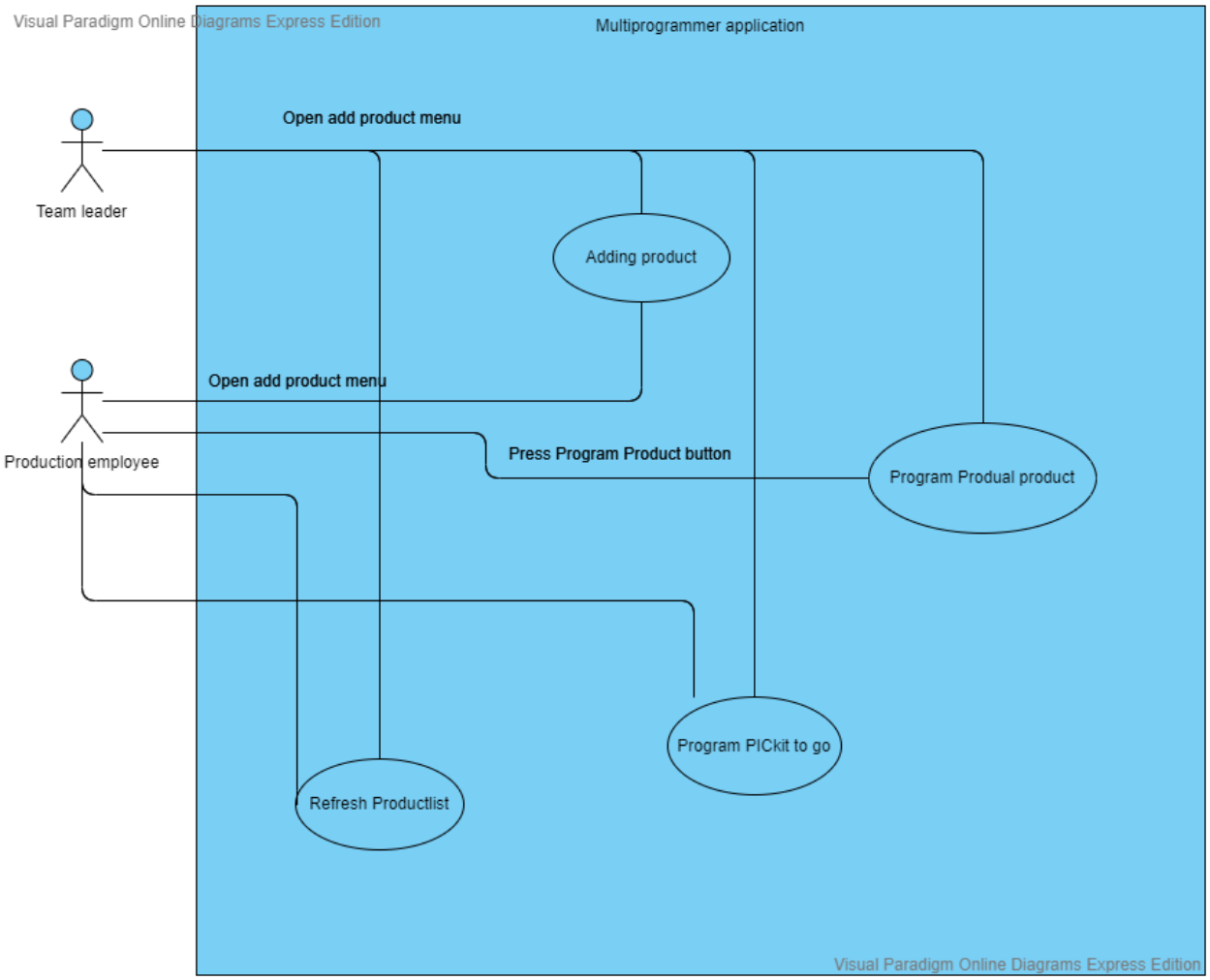

Figure 2: Commissioner's application use case model.

Use cases can also be described in a separate document called a use case document. The use case document consists of the following elements (Project Management Docs):

- The name of the use case.
- A description of the reason for the use case and expected outcome.
- Actors, i.e people who interact with the use case. These can be primary, i.e they are the ones who initiate the use case, or secondary, i.e those who participate in its completion.
- Preconditions that must be fulfilled before the use case can be executed.
- Postconditions that describe the system status after the use case is complete. Postconditions can also describe what happens if the use case is unsuccessfully executed.
- Flow, or a sequential description about the normal flow of the use case's functions.
- Alternative flow, or special conditions that are not part of the main flow. They are the result of exceptions in the primary flow.
- Exceptions, describing any errors that can occur during the execution of the use case.
- Requirements, or descriptions of non-functional or special requirements that are needed for the use cases execution.

An example the contents of a use case document (Figure 3):

Name of Use Case:	Order Materials		
Created By:	ABC Corporations	Last Updated By:	J. Doe
Date Created:	02/15/xx	Last Revision Date:	02/22/xx
Description:	ABC Corp. buyer submits material order to one of a pre-approved list of material vendors		
Actors:	ABC Corp. buyer, SAP material module, pre-approved vendor		
Preconditions:	1. Vendor has pre-approval in ABC Corp.'s ordering system 2. Funding is available for material ordering 3. Material being ordered is available for purchase		
Postconditions:	1. Vendor receives funds for purchase of materials 2. ABC Corp. receives materials within the designated timeframe 3. ABC Corp.'s material account is reduced by the cost of the material order 4. ABC Corp.'s inventory numbers are successfully updated once material is received		
Flow:	1. ABC Corp. buyer identifies material needing to be ordered 2. ABC Corp. buyer consults pre-approved list of vendors to identify supplier 3. ABC Corp. buyer confirms funding is available 4. ABC Corp. buyer submits order to pre-approved vendor 5. Vendor receives order and verifies material is available and accepts funding transfer 6. Vendor pulls material order and submits shipping order to ship material 7. ABC Corp. receives material 8. ABC Corp. enters material receipt verification into SAP and inventory levels are updated 9. Funding transactions are confirmed between ABC Corp. buyer and vendor		
Alternative Flows:	5. In step 5 of the normal flow, if the vendor does not have the material available 1. Vendor places order in a hold status and notifies the ABC Corp. buyer 2. Vendor provides updates and estimated timeframe of material receipt 3. Once material arrives the Use Case resumes at step 6 of the normal flow		
Exceptions:	2. In step 2 of the normal flow, if ABC Corp. identifies material needed with no pre-approved vendor 1. ABC Corp. buyer initiates internal process to identify suppliers for new material 2. ABC Corp. buyer coordinates agreement between ABC Corp. and potential vendor 3. Upon obtaining agreement and approval, vendor is added to pre-approved vendor list 4. Use Case resumes on step 3 of normal flow		
Requirements:	The following requirements must be met before execution of the use case 1. Funding availability must be verified prior to submitting any material purchases 2. All material orders must comply with internal ABC Corp. ordering guidelines and procedures		

Figure 3: Use case sample document (Project Management Docs).

The final product of the requirements specification stage of the software engineering process is the software requirements specification document or in short SRS. This aims to provide developers with a detailed description about the system that is to be implemented. When the software is developed outside of one's own company, the document should be as detailed as possible as this lessens the chance of errors being made during the outsourced implementation. However, it can be less detailed if the development for the software is done within one's own company. (Sommerville 2016, 126-127.) One example of the contents of the software requirements specification document that is based on IEEE 1998 standard for requirements is provided by Sommerville (Figure 4).

Chapter	Description
Preface	This defines the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This describes the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This defines the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The nonfunctional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter presents a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.
System requirements specification	This describes the functional and nonfunctional requirements in more detail. If necessary, further detail may also be added to the nonfunctional requirements. Interfaces to other systems may be defined.
System models	This chapter includes graphical system models showing the relationships between the system components and the system and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This describes the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These provide detailed, specific information that is related to the application being developed—for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

Figure 4: SRS document contents (Sommerville 2016, 128).

The requirements specification stage for the software engineering process ends with the validation of the software requirements. The validation of the requirements simply means that the developer confirms with the client that the requirements define the kind of software that the client wants (Sommerville 2016, 129).

Different kinds of check procedures for the requirements can take place at the validation stage. These include checking that the requirements meet the needs of the client, they do not contradict each other, the requirements document includes all requirements and constraints for system functions, the requirements can actually be implemented, and finally, that tests can be written for the requirement. There are also techniques that can be used to validate the requirements. These include analyzing the requirements for errors, creating prototypes of the software, and seeing if it is possible to create test cases for the software. (Sommerville 2016, 129.)

2.2 Design

In the design phase, the requirements created in the requirements specification stage are used to create documentation that the developer can use to implement the requirements (Cooling 2019). In this stage, the individual components of the software and their relations and interaction with each other are defined (Sommerville 2016, 197).

Depending on the scope of the software, the importance of this stage varies. For smaller applications, this stage and the implementation stage are the only actual stages that are used in the creation of the software while larger applications tend to include all the other steps as well. (Sommerville 2016, 197). In general, like the requirements specification stage, the design stage consists of multiple different types of design procedures that can take place. These are as follows: interface design, architectural or large-scale design and detailed design (Bell 2005, 23). The end output of the design process is called SDD or software design document aka software design description. The SDD provides a model of the software that is being developed (IEEE-SA Standards Board). In the SDD, the software is split to different components, and the relationships between the components are defined. The SDD should consist of at least the following parts (IEEE-SA Standards Board):

- An introduction
- Decomposition description that describes the software components and their functions.
- Dependency description where the relationships between the components are described
- Interface description which provides designers, programmers and testers with information on how to use the different functions of the software component
- Detailed design description which contains the information that programmers need before implementation

An example of the contents of an SDD (Figure 5):

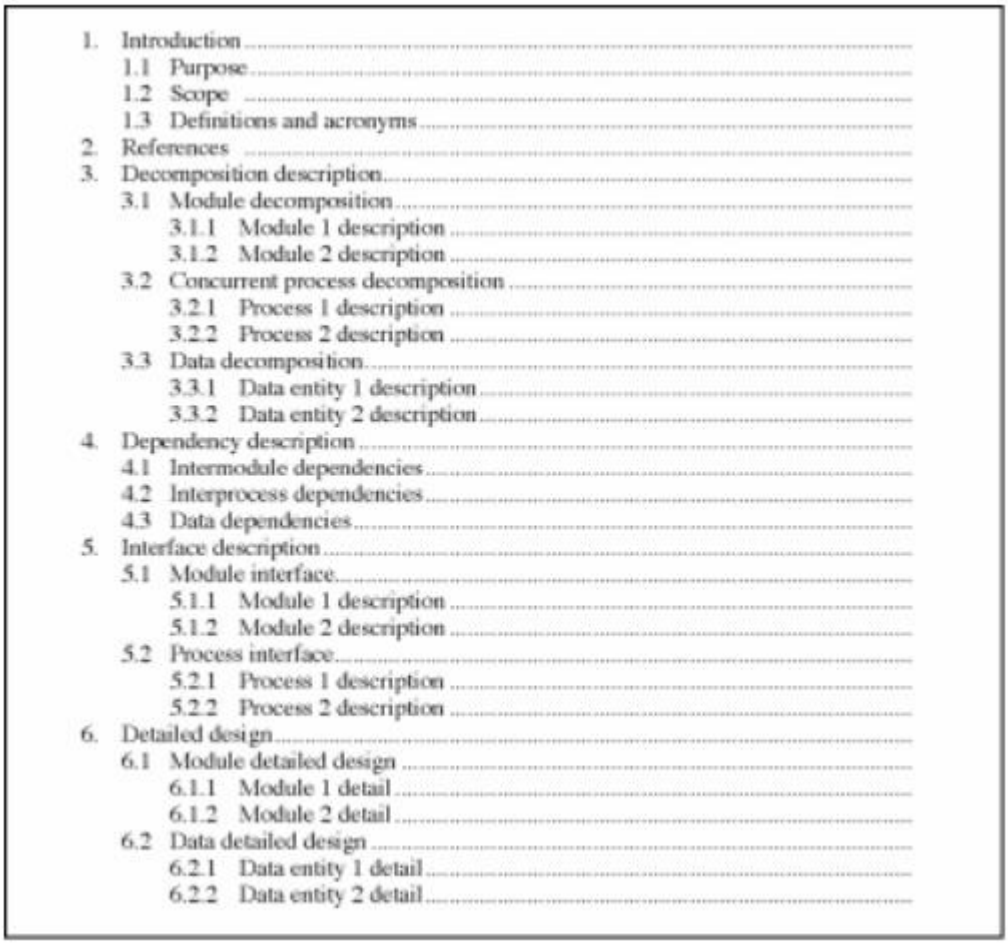

Figure 5: SDD table of contents (IEEE-SA Standards Board).

Sommerville (2016, 199-201) introduces a stage of the design process that can come before the others and calls it system context and interactions design. During this stage, the developer creates models that define how the software interacts with its environment. The two types of models that can be created at this stage are:

- System context models that describe other systems that exist in the environment for which the software is developed.
- Interaction models that show how the software interacts with its operating environment. One interaction model type is a use case model that is described in chapter 2.1.

Sommerville (2016, 201) also recommends that the models should not contain too many details.

In the architectural design stage, the developer designs as the name suggests the architecture of the software that is to be implemented. During this stage of the developmental process, the software that is to be implemented is broken down into individual components, and the interaction between them are defined. (Sommerville 2016, 201.) A tool that can be used during the architectural design stage is a class diagram as it shows the different components and their interactions (Figure 6)

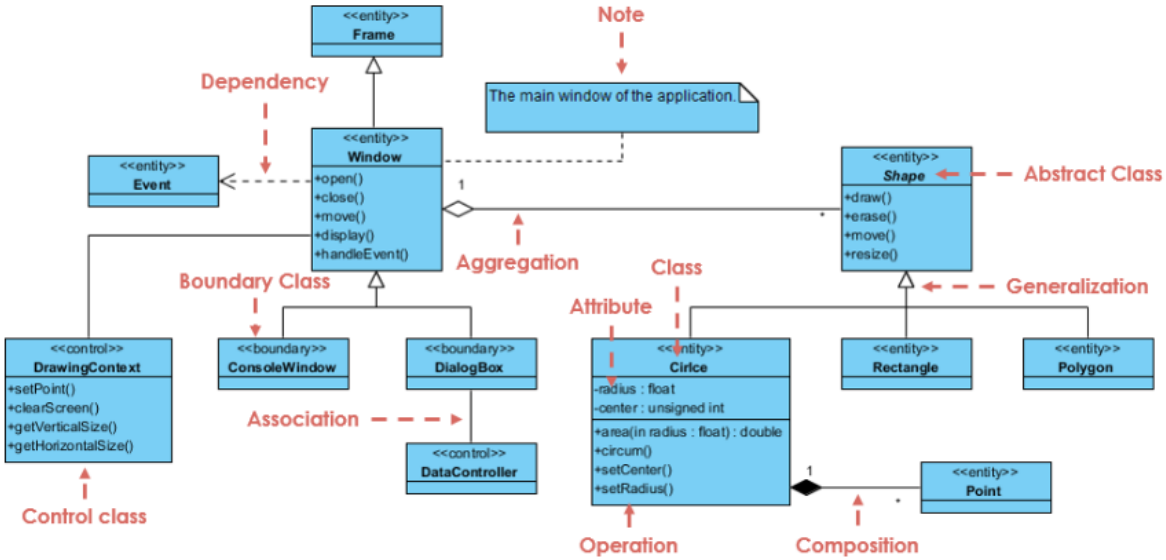

Figure 6: Class diagram example (Visual Paradigm no date).

Database design is a design process step that could be considered part of architectural design. In this step, the database for the software is designed, including the software's data structure and its representation. This step is not always necessary as it depends on whether the software uses an existing database or if it requires the creation of a new one. (Sommerville 2016, 57.) An entity relationship diagram is a good tool for database design as it provides a visual aid concerning the different entities in the software that contain data and how they relate to each other. An example of an entity relationship diagram is shown in Figure 7.

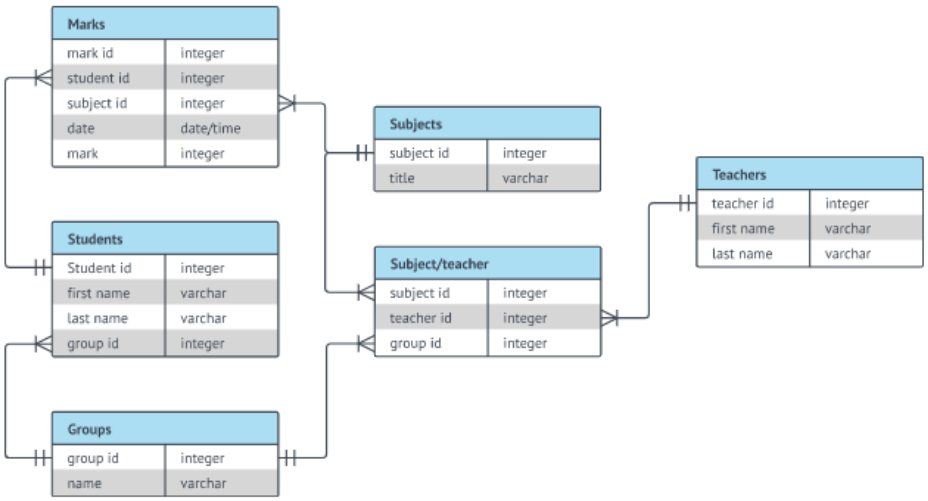

Figure 7: ER-diagram example (Lucidchart no date).

In the interface design stage, the user interface is designed to be as easily usable as possible for the end user (Bell 2005, 53). There are three types of interfaces that a program can use (Bell 2005, 54-55):

- Command line interface that runs commands through the command line.
- Menu interfaces that use choices to run commands. For example, an ATM uses a menu interface.
- Graphical User Interface (GUI) that directs the software through an interactive interface where selections are made using a mouse.

When designing the user interface, the developer should keep in mind the different kinds of users that might use the software and as such the interface should be designed in a way that accounts for the differences between users. Alternatively, an interface can be created from the start to be used in different ways (Bell 2005, 56). In his book, Bell (2005, 57) describes three principles that should be kept in mind during the designof the user interface:

- Learnability - the interface is easy to learn how to use.
- Flexibility - the interface must account for different ways of using it.
- Robustness – the interface must properly inform the user about what is going on.

This design stage can also include prototyping the interface to see if it meets the requirements set for the interface (Bell 2005, 64). The detailed design stage involves creating the detailed designs for the different modules and components of the software (Bell 2005, 24).

2.3 Implementation

The implementation stage is a vital part of the software engineering process as it is during this phase that the software is actually programmed. The implementation stage is linked deeply with the requirement specification and design stages as they provide the framework upon which the developer starts to create the software. If the previous stages have provided output that follows most of the practices discussed in the respective chapters of this thesis, the developers' job becomes much easier as they do not have to spend time seeking clarification about the clients' intentions. The implementation is based around documents and outputs made in the previous steps, such as the use case diagrams and documents, class diagrams and design documents.

According to Sommerville (2016, 212), there are three aspects of the implementation stage that are not usually covered in programming texts:

1. Reuse. The developer should use as much existing code as possible.
2. Configuration management. The developer should keep track of the versions of the components created during the development process, so the wrong component version is not used during development.
3. Host-target development. The software is developed on one computer (host system) and the executable file is run on another computer (target system).

There are numerous different ways to approach the programming of the software. In this thesis, two approaches are examined: object-oriented and structured. In the object-oriented approach to programming, methods and variables that are linked to each other are encapsulated together in a class. Classes themselves are in essence abstractions which allow the developer to reuse pieces of code. In an object-oriented approach, variables are, in general, private, and methods are public. The user interacts with the methods, and the methods interact with the variables. (Bell 2005, 200-208; Kent 2014, 169.) A key component of object-oriented programming is the concept of inheritance. If the objects that are being created are very similar, but one is a specialized case of the other, the former has inherited from the latter. In his book, Kent (2014, 169) provides an example of how inheritance works: a class called

RawTurtle (Figure 8) was created first and then a subclass called Turtle was created (Figure 9). Turtle inherits the attributes from the RawTurtle class.

```
class RawTurtle(TPen, TNavigator):
    """Animation part of the RawTurtle.
    Puts RawTurtle upon a TurtleScreen and provides tools for
    its animation.
    """
    screens = []

    def __init__(self, canvas=None,
                 shape=_CFG["shape"],
                 undobuffersize=_CFG["undobuffersize"],
                 visible=_CFG["visible"]):
```

Figure 8: Main class (Kent 2014, 170).

```
class Turtle(RawTurtle):
    """RawTurtle auto-creating (scrolled) canvas.

    When a Turtle object is created or a function derived
    from some Turtle method is called a TurtleScreen
    object is automatically created.
    """
    _pen = None
    _screen = None

    def __init__(self,
                 shape=_CFG["shape"],
                 undobuffersize=_CFG["undobuffersize"],
                 visible=_CFG["visible"]):
        if Turtle._screen is None:
            Turtle._screen = Screen()
        RawTurtle.__init__(self, Turtle._screen,
                           shape=shape,
                           undobuffersize=undobuffersize,
                           visible=visible)
```

Figure 9: Subclass (Kent 2014, 169).

In the structured approach to programming, it is said that programs are composed of only three components (Bell 2005, 88):

- Sequences i.e the program instructions which are written in the order they are executed.
- Selections. If-then-else program statements.
- Repetitions. While-do program statements.

In the structured approach, a program has only one start and exit point and it can consist of multiple components that follow this approach (Bell 2005, 88; Bell 2005, 97). In addition to these two approaches, there are many other ways to program a software product.

2.4 Testing

Testing has two main purposes in the software engineering process. The first is to confirm that the requirements set during the requirements specification stage are satisfied (validation testing), and the second is to finds flaws, or bugs that stop the software from functioning as specified (defect testing) (Sommerville 2016, 227). In addition, a software product also tends to go through the following types of testing:

- Development testing where the software is tested during development
- Release testing where the completed version of the software is tested
- User testing where the users test the software in their own environment. (Sommerville 2016, 231.)

As mentioned before, development testing takes place during the development process of the software and has three separate stages. The first of these testing stages is unit testing. In unit testing, the tester focuses on testing the software's individual components and functions. (Sommerville 2016, 233.)

The second type of development testing is integration testing in which numerous software components are integrated to form complete systems and tested. There are three different ways to approach integration testing. The first is the big bang approach, in which all the parts of the final system are tested without prior unit testing. This approach presents the risk of not being able to locate the component causing an error in the software. The second approach involves first doing unit testing and then performing the big bang testing. The third approach to integration testing is incremental testing. In this approach, a component of the software is tested before another component is linked to it and this is repeated until all the components have been tested. This approach provides a way to find faults in the components more easily. Since the components are combined one at a time, the latest of the added components is the most likely cause of any faults. (Bell 2005, 277.)

The last type of development testing is system testing where the entirety of the software system is tested. The purpose of system testing also tends to be to detect defects within the program (Sommerville 2016, 245).

Release or acceptance testing is rather like system testing, although with some key differences. During release testing, a completed version of the software is tested instead of a version that is still in development. Also, during release testing, the software is tested to see that it meets all the users' requirements while system testing tends to test simply the general functionality of the program.

The last of the main testing phases is user testing. User testing tends to take place during system testing. During this phase, the intended users of the software test it in their own environments. User testing is an important part of the development process as even if the software development and release testing were performed, the users' actions and operating environments can cause behaviors that were not detected before or identify bugs that were not found during the preceding stages. (Sommerville 2016, 249.) Sommerville (2016, 249) describes three different types of user tests that can be performed:

- Alpha testing where the users test the software during development, so this type of testing overlaps with development testing. Alpha testing tends to be performed by only a selected group of people.
- Beta testing where the software is released for a wider audience and their feedback on possible flaws or bugs is gathered.
- Acceptance testing which is the final testing before the software is released. During this stage, it is determined if the software meets all set requirements and is in acceptable condition.

An important aspect of testing is the V-model of software development. It is also known as a plan-based software process. In the V-model, there are testing related tasks for all the stages of software development except for the implementation stage (Figure 10).

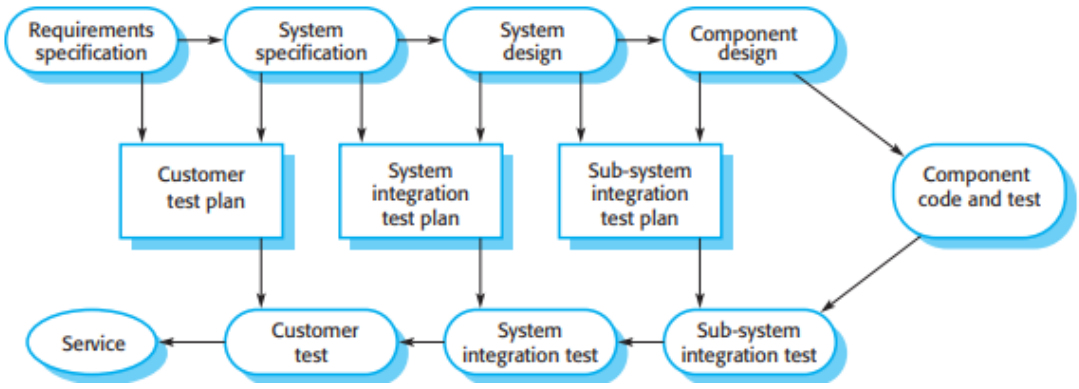

Figure 10: V-model example (Sommerville 2016, 60).

During the requirements specification and design stages, this entails the creation of test plans for the integration and user test phases. If the V-model is used, these plans are what direct the testing of the software. (Sommerville 2016, 59-60.) Testing also involves the creation of test cases which are written specifications about what kind of input should be given to the program and what kind of output is expected. The general process of testing which is followed from development testing to user testing begins with the creation of test cases and ends in a test report. While test cases cannot be generated automatically, the execution of the test cases can be automated as seen in the testing process flowchart in Figure 11. (Sommerville 2016, 230; Figure 11.) Automated testing can save a great amount of resources and time that can then be allocated to other matters.

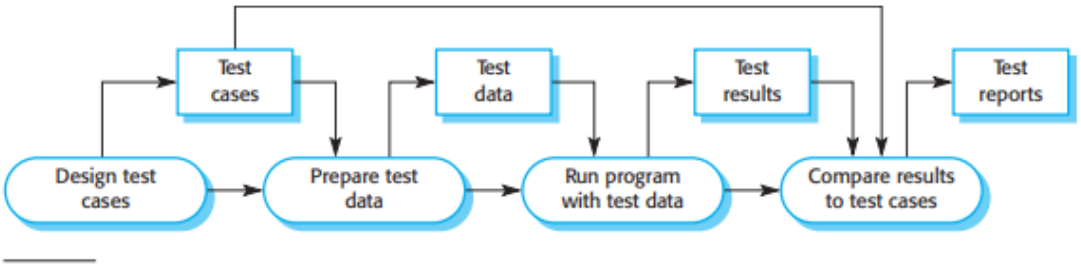

Figure 11: Testing process flow (Sommerville 2016, 230).

2.5 Deployment and maintenance

The deployment stage is the phase of the development process where the completed software is released to its users. When the software has been completed and it has passed the acceptance tests and the users, developers and testers so agree, the software is released for its intended users.

The maintenance stage takes place after the product is released. It encompasses actions that modify the software or help maintain its functionality. There are two types of maintenance that are used. The first type is remedial maintenance which focuses on fixing the flaws found in the software after its release. The second is adaptive maintenance that involves either adding new features to the software or modifying the current software to account for changes in its operating environment, such as operating system and hardware. (Bell 2005, 11.)

In general, maintenance tends to be the most expensive part of the software development process. As seen in Figure 12, the cost of maintenance can take well over half of the total costs of the entire development cycle, and Figure 13 shows that the addition or modification of features is the costliest of all maintenance processes.

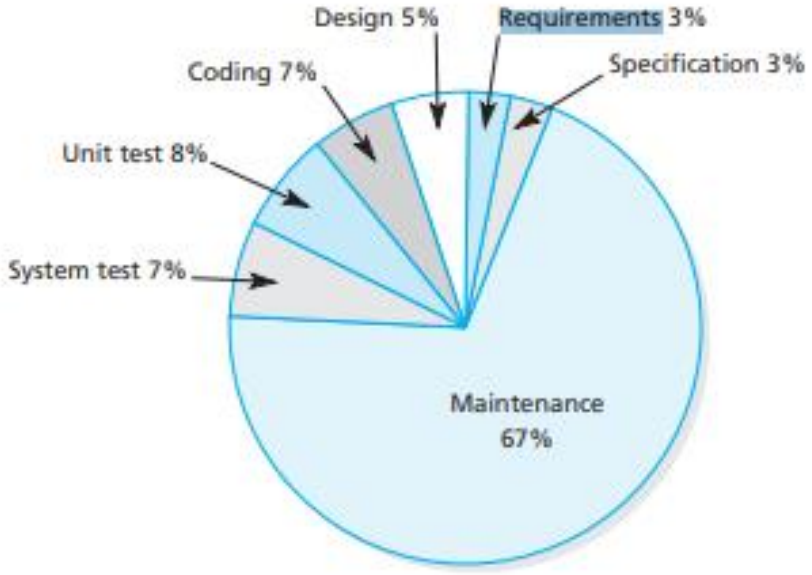

Figure 12: Software development cost graph (Bell 2005, 12).

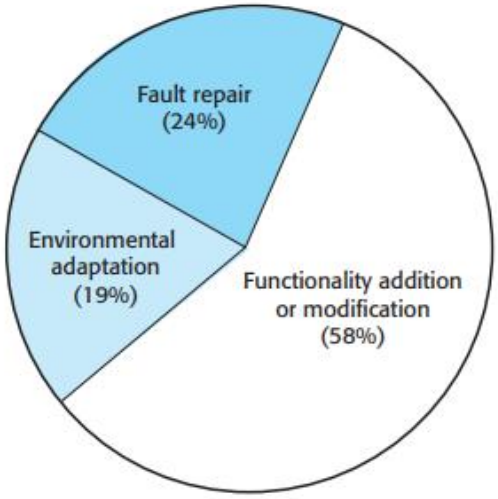

Figure 13: Maintenance cost distribution (Sommerville 2016, 272).

There are numerous factors that cause the cost of maintenance to rise to such high levels. One factor is the age of the software. This is due to the fact that any changes made to the software are likely to change the structure of its code which in turn makes it harder to understand and modify. Another reason is that if maintenance and development duties are separated between different parties, the developers have no incentive to make their code maintainable

since they will not be responsible for its maintenance. If people not involved in the development of the software are assigned to maintain it, they must put in time and effort to understand the software before they can properly do anything with it. This also leads to increased maintenance costs. (Sommerville 2016, 272-273.)

2.6 Use of software engineering principles in the commissioner's application

The commissioner's application did not follow all the different principles described in the previous chapters. This is largely due to the software being much smaller in scale compared to large-scale software, which tend to follow the principles more closely. The use of the principles in the creation of this particular application is described below stage by stage.

The requirements specification stage was short due to the limited functionality of the software. The requirements specification was mostly made during a meeting with representatives from Produal. Their desired requirements for the application were documented in a text file to serve as as a basis for the actual specification.

The design stage mostly entailed the making of a general outline concerning what different components each of the functional requirements should contain and how they could be created. This allowed the application to have a general architectural design outline. The details of the design outline became more specified as progress was made with the development process. The design process also contained the interface design phase as the commissioner's application contains a GUI. The interface design phase was conducted by prototyping the GUI and refining its visual appearance in meetings with the team leaders from Produal.

The implementation stage followed the structured programming approach. The implementation consisted of different classes which contained the different application functions. Thus, the program is object-oriented, even if it also features elements of modular programming due to the functions basically being subroutines for specific tasks.

During the testing phase of the application, the three main testing types were all involved. The development testing mostly took the form of component testing due to the application's programming functionalities relying upon the GUI. More specifically, the primary function takes the form of a command line argument that must be based on the choices made in the GUI. The system testing showed no obvious flaws. The release and acceptance testing was done in Produal's production environment, and the tests showed some performance and GUI issues which made it imperative that the application was returned to the implementation stage to further optimize the program and fix the issues found with the GUI. After the identified issues were fixed, the application was deemed to meet the requirements and acceptable for deployment. A user manual was made during the deployment phase and it is presented in Appendix 1.

In general, the operating procedure of the waterfall model was not strictly followed as some parts of the process overlapped with others. For instance, parts of the requirement specification process were done during the implementation stage due to the client wanting new features that were not discussed during the original meetings where the requirements were defined.

The program was released for use in Produal's production environment, and the application was installed on some of their computers. The application is currently in use at Produal and has been moved to the maintenance phase. A maintenance manual is currently under construction.

3 TOOLS AND TECHNOLOGIES

This chapter describes the tools and technologies used in the creation of the commissioner's application. Both physical hardware tools and different kinds of software languages and libraries were used in the creation of the application.

3.1 PICkit debuggers and programmers

PICkit is a series of physical in-circuit debugger and programmer products produced by Microchip Technology Inc for the purpose of programming microcontrollers (Microchip 2020b). The commissioner's application supports two types of PICkits, PICkit 3 and PICkit 4.

PICkit devices are powered through an USB port, and in the case of PICkit 4 they can be powered from the target device i.e.the microcontroller that the PICkit programs. PICkits 3 and 4 have two modes for programming PICkits. The first mode allows programming the target device directly through a computer by using the MPLAB X IDE or IPE. The second mode involves the setting of the PICkit to Programmer-to-Go mode. In order to set the PICkit to Programmer-to-Go mode either the IDE or IPE is used to create a device memory image that is then uploaded to the device's storage, in PICkit 3 to internal memory and in PICkit 4 to an inserted MicroSDHC card. Files that are used to create the image are usually HEX files. If the PICkit is in Programmer to Go-mode, there is no need to use the IDE or IPE to program the target device since in Programmer to Go-mode all the needed information is stored in the PICkit. (Microchip 2020b.) However, one PICkit can only store one image at a time.

3.2 MPLAB X IDE, IPE and IPECMD

MPLAB IDE X is an IDE (Integrated Development Environment) specifically tailored to allow its user to control, debug and develop applications for Microchip's embedded controllers i.e. controllers designed for one task (Microchip 2019). MPLAB IPE is used for directing a PICkit device's programming functions, and in contrast is often used in a production setting (MPLAB X IDE User's Guide 2020). IPECMD is a command line interface that runs the same programming functions as the IPE through the computer's command line.

3.3 Python

Python is an interpreted, object-oriented and high level programming language featuring high-level built-in data structures as well as dynamic typing

and binding that provide a platform for rapid application development. The easy readability of its code reduces the maintenance costs. (What is Python.)

Python also provides developers with numerous advantages as described by Parikh (2018), for example:

- The simple syntax makes it simpler to learn than some other languages.
- Comparatively lower line count allows to perform the same tasks with less effort. For example, compared to C++, a task that takes 7 lines can be done in 3 lines in Python.
- Python's large community gives access to ready libraries for almost any kinds of tasks that need to be performed by a program.

Python was used as the exclusive programming language for the commissioner's application due to the large amount of online documentation regarding various problems that might be faced, easy syntax and the ability to quickly write code for prototype features and the extensive amount of libraries for the tasks the application must be able to perform.

3.4 Kivy

Kivy is a graphical user interface library for Python. It allows the developer to make user interfaces for many different platforms. Kivy is currently the only viable way to make interfaces in Python for Android (Phillips 2014, 1).

Kivy has its own language syntax called **`kvlang`** that allows the user to rapidly design interfaces for an application by simply writing it to a separate **`kv`** file (Phillips 2014). **`Kvlang`** is an excellent tool for rapid deployment of GUIs as its syntax is close to standard written English and thus has a minimal learning curve. **`Kvlang`** also makes the code easier to manage since all data required by the GUI is stored in one sub-file which can be called from the main file. Phillips (2014) shows the following example on how to bring the classical "Hello World" message to the screen (Listing 1):

```
Label:    text: "Hello World"
```

Listing 1: Kvlang "Hello World" example (Phillips 2014, 7)

Kivy has numerous widgets that can be used, such as labels, buttons and spinners. The attributes of the widgets such as size, position or color can be modified freely by the user. A developer can also use **`kvlang`** to create their own custom widgets by combining existing widgets. (Phillips 2014, 9-10.)

4 IMPLEMENTATION

In this chapter, the implementation of the different parts of the application is described, and explanations and reasons for the choice of solutions are presented. In the subchapters, when individual keywords from the source code in Appendix 2 are referenced, they are identified in boldened Consolas font, size 12.

4.1 The GUI (Graphical User Interface)

The development process for the application began with the design and creation of the GUI as the other features are dependent on the elements contained in the GUI.

The GUI was developed in cooperation with the production team leaders. The first design was created with the use of draw.io and presented to the team leaders to showcase the general visual appearance of the GUI and the placement of the elements contained within the GUI (Figure 14).

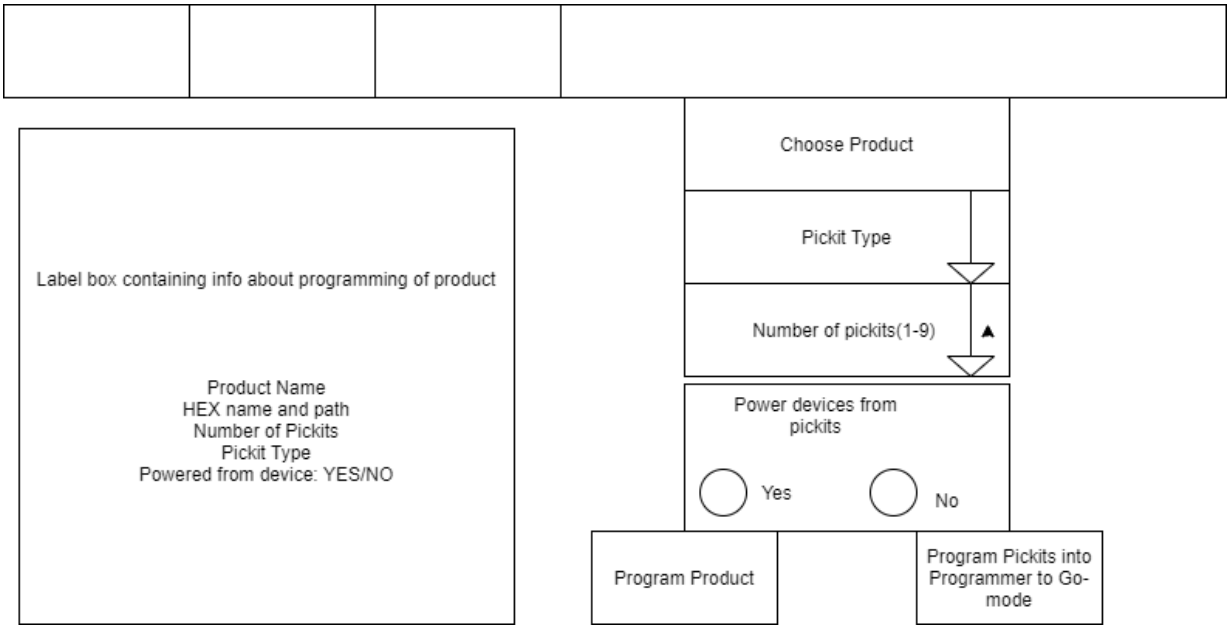

Figure 14: First concept of the GUI.

The GUI was further developed based on feedback gathered from subsequent follow-up meetings with the team leaders until it the iteration seen in the final version was achieved. The largest change made in the process was that the choice for the amount of power the PICkits were to supply to the target device was removed, as it was deemed better if the power setting was included in the configuration file so that there would be no need to select it every time as the products always use the same voltage when they are being programmed.

The final version of the GUI is presented in Figure 15. For the final version, the “Add Product” and “Refresh the product list” buttons were added because

the scope of the functionalities was increased later in the developmental cycle.

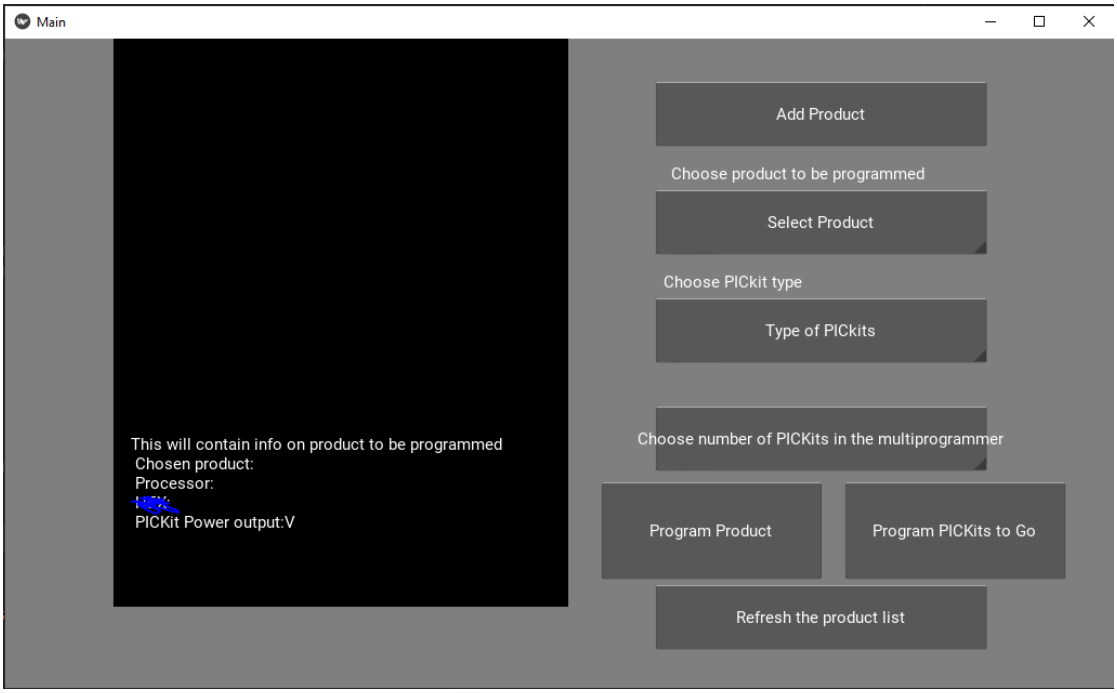

Figure 15: Final GUI appearance.

In general, the GUI for the application consists of Spinner and Button widgets from the Kivy library. The Spinners were chosen for the widgets used to display the lists of products and other options because their values could easily be changed, which offers a far quicker way to create a dropdown-style selection method than Kivy's own Dropdown widget. The Dropdown widget is a better option if a developer needs different widgets in the dropdown menu. For the commissioner's application, only the ability to choose from different options was needed, a requirement which the Spinner widget met perfectly.

Kivy has multiple different layouts that can be used to arrange the GUI elements. The different screens for the application and their contents are defined in their separate classes, each having their own layouts, which are defined as the parent class for the screen. For example, the main menu could be defined like this: **`class MainLayout(FloatLayout)`** (Appendix 2). For the main menu, the **`FloatLayout`** was used since it allows the different GUI elements to be freely moved around the screen. The different elements for the GUI are initialized in the init function of the class in the following manner (Listing 2):

```
self.deviceprogram = Button(text="Program Product",
                            size_hint=(.2, .15),
                            pos=(550, 100))
```

Listing 2: Initialization of a GUI element (Appendix 2).

As seen from the above example, the widgets are initialized by defining their text, size and position. The self-keyword seen before the widget's name refers to an instance of the class, in this case **`MainLayout`**, and is used to declare parts of the class (Parker 2016, 222). After the different attributes of the widgets are defined, they are added to the layout using Kivy's **`addwidget`** command that takes the widget as an argument. For example, the "Program Product"-button seen before is added with:

```
self.add_widget(self.deviceprogram)
```

Listing 3: Adding the GUI element (Appendix 2)

The main menu's GUI also gives the user information about the product that is about to be programmed by using Python's Configparser library in the **`updatelabel`** function. The **`configparser`** is initialized in the **`MainLayout`** class. (Appendix 2.) The **`updatelabel`** function opens the ProductList.ini configuration file with the Configparser and reads the text from the currently selected element from the Choose Product Spinner (Appendix 2). As the "Choose Product" Spinner's values are taken directly from the section names in the configuration file, the **`updatelabel`** function reads the product's processor type and power settings from the section corresponding to the product's name. (Listing 4)

```
self.parser.read("ProductList.ini")
produalprocessor = self.parser.get(self.produalproduct.text, 'Processor')
produalpower = self.parser.get(self.produalproduct.text, 'Power')
produalhex = self.locatehex()
self.infobox.text = 'This will contain info on product to be programmed\n Chosen product:%s \n
Processor:%s \n HEX:%s\n PICKit Power output:%sV\n' % (self.produalproduct.text,
produalprocessor, produalhex, produalpower)
```

Listing 4: Updatelabel functionality (Appendix 2).

The **`locatehex`** function (Appendix 2) presented in Listing 4 uses Python's glob library in conjunction with the Configparser to locate the file for the chosen product. In a similar fashion to the **`updatelabel`** function, the parser is used to find the defined **`filepath`** in the configuration file.

This file path is then used as an argument for the **glob** function (Appendix 2) of the glob library to locate all the files that match the defined file path. Because the glob library allows for the use of wildcard characters, the user can leave parts of the path open by using *. For instance, should the defined path be C://file*.png, the **glob** function would locate all PNG files with file in their name in the root C directory. The glob library was chosen because it allows the use of wildcard characters to help account for any future updates, since the path in the configuration file can simply use the * character to mark any parts of the filename that are likely to change when the product's program is updated. As the **glob** function creates an array of all matches (glob – Filename pattern matching), the locate hex function sorts the matches by modification date and uses the most recent one.

For the other screens in the application, Kivy's Popup widget was used because it allows the creation of other screens without having to use Kivy's Screenmanager functionality to handle the transitions between the screens. The use of separate screens was considered unnecessary since, after all, the other screens are only used for a short moment. The **Popup** function creates a popup screen that contains the layout and elements as defined by the user.

4.2 Programming a Produal product and programming PICkit

The main functionality of the application is to program Produal's products by calling IPECMD through the command line using Python's sub-process library. The programming functionality begins after the user has made all the choices in the main menu. After the user presses the "Program Product" button, the program calls the **run_all** function that first calls the popup function to create a Popup that contains the progress bar and the buttons for programming a product and returning to the main menu. (Appendix 2.) After the popup is called, **run_all** defines the length of the progress bar to be equal to the chosen number of PICkits multiplied by 100 (Listing 5).

```
maxprogress = int(self.pickitnumber.text) * 100
self.p.max = maxprogress
```

Listing 5: Setting the progress bar maximum value (Appendix 2)

An array called **`threads`** is created in **`run_all`** to store all the separate calls to the programming function. Using a for-loop, separate threads for each number in the range of 1 to the chosen number of PICkits are created and stored in the array. The **`thread`** operator sends the current number in the loop and **`maxprogress`** as arguments to the programmer function. Later in this chapter, the current number in the loop will be referred to as CNL (Current number in loop). (Appendix 2.) The end of the range is defined as the chosen number of PICkits that is read from the corresponding GUI element plus one. The reason 1 is added to the number is due to the way Python's range function works. The range is calculated until it is the final value minus 1. For example, if a range from 1 to 4 is defined, the actual numbers calculated are 1, 2 and 3. (Python range () no date.) Threads were used to allow the different calls to the programming function to be run as concurrently as possible.

After the array is filled with the correct number of threads, each of the threads is started using a for loop. The for loop ensures that no other threads are running before starting a new thread. This leads to the functionality of the programmer function. The function first uses the Configparser (Chapter 4.1) to read the chosen product's processor type and power settings while adding a message about which PICkit is running the function by utilizing the CNL sent from the run_all function (Figure 16).

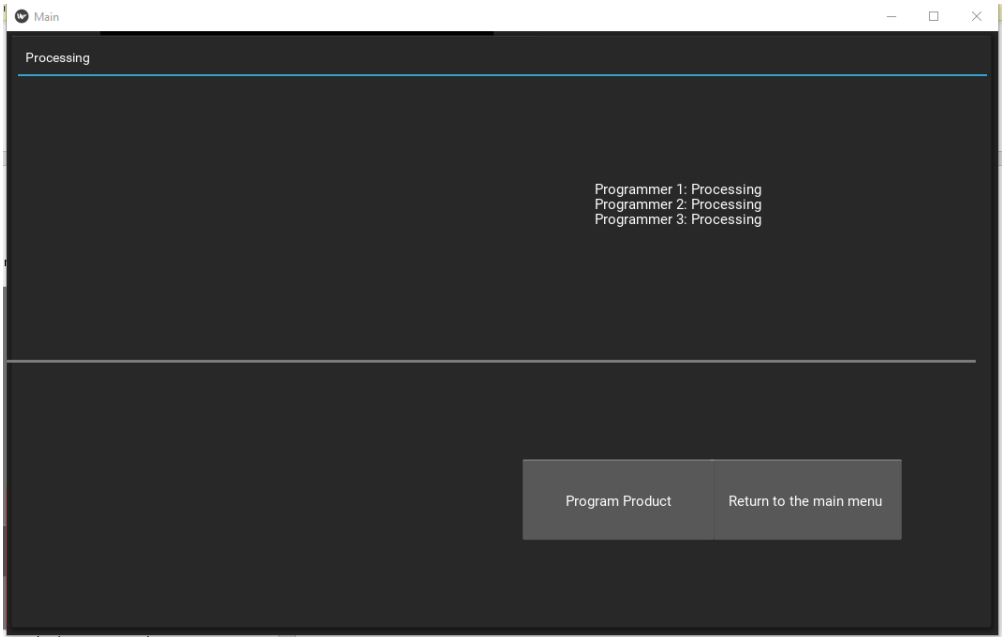

Figure 16: Processing message.

The actual programming occurs in the **`run_command`** function by using the **`Popen`** functionality of Python's subprocess library. **`Popen`** was chosen instead of the other functions that could be used to run commands due to its communication function. The command is supplied to **`subprocess.Popen`** as a string. (Appendix 2.) The command that IPECMD uses to program a product is created by calling **`"ipecmd.exe"`** from the computer (Appendix 2) and then adding the necessary arguments by using the Configparser to read the product's information from the correct section in the configuration file. The **`locatehex`** function locates the file required by the product and supplies its path as a variable (Appendix 2). These arguments are then added to the **`subprocess.Popen`** call by utilizing Python's line formatting that allows variables to be included in strings by using % as an operator to assign a variable type such as %s for string or %i for integer (String Formatting no date). Additional operators are added to the **`Popen`** call, and **`universal_newlines`** (Appendix 2) is set to true so that the output would be properly formatted instead of being printed as a single string line.

If there are more than a single PICkit connected to the computer, IPECMD asks the user to input the index number of the PICkit they want to use to program the target device. The CNL is used as the index number, and the input is sent to **`Popen`** using the function call **`subprocess.communicate(input=CNL)`** which waits for the command to finish before sending the input. (Appendix 2.)

After the input is sent, the PICkit begins programming the device. The continued output is captured by using **`subprocess.communicate[0]`** (Appendix 2) to read only the output as **`subprocess.communicate`** on its own sends a tuple containing both the output and possible error data (subprocess — Subprocess management no date). The output is then read, and if it contains a message about the programming succeeding or failing (Figures 17 and 18), the message is displayed on the screen, and **`return`** is used to cause the function to finish. The value 100 is added to the progress bar which is then shown on screen as the progress bar filling.

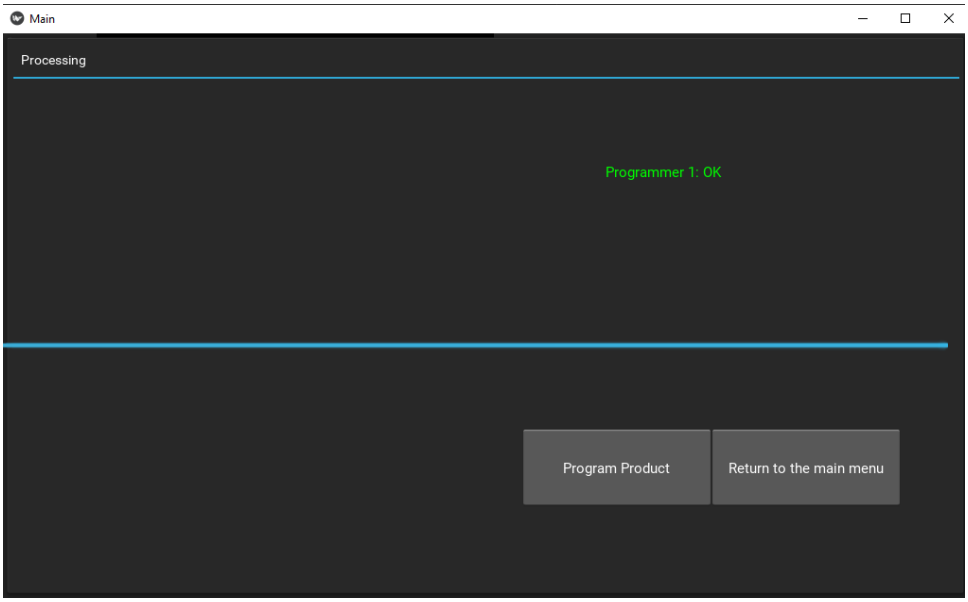

Figure 17: Programming successful.

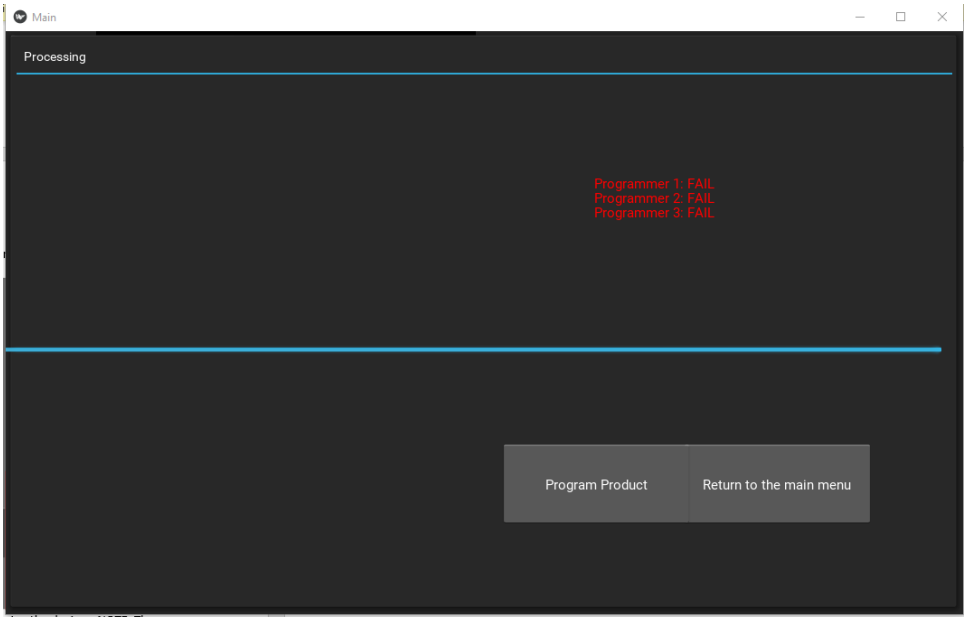

Figure 18: Programming failed.

The process of programming the connected PICkit to the Programmer to Go-mode follows the same steps, with only the arguments given to **`subprocess.Popen`** being different. Currently, only PICkit 3s can be set to Programmer to Go mode because Microchip Technologies do not allow to program PICkit 4s to Programmer to Go-mode in IPECMD's current version.

4.3 Adding a new product

The functionality of adding a new product allows the user to add a new product to the **`productlist.ini`** (Appendix 2) configuration file. Each computer has its own configuration file, and in the future the aim is to have the configuration file available from a cloud service or SVN so the same file would be available for all users.

The process of adding a new product to the configuration file begins by pressing the "Add product" button from the main menu which opens a Popup widget containing the necessary fields set in a **`GridLayout`** with two columns and four rows (Figure 19).

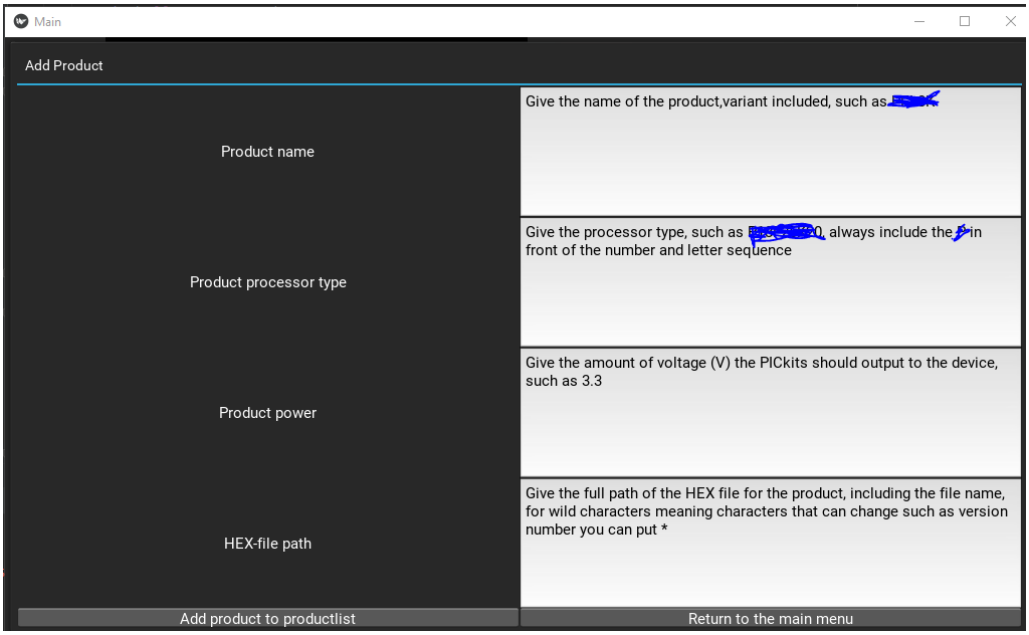

Figure 19: Add product-menu.

The information that is transferred to the configuration file is taken from the text fields on the right side of the screen. Instructions for the user on how to give the proper data are given in the the default values of the text fields. The user replaces the text in the text fields with the proper data.

After the “Add product to productlist” button is pressed, the **`write_product_info`** function reads the text field values. Using the Configparser library, the function creates a new section under the product name and adds the rest of the given data as options for the section. The **`configparser`** is then used to open the configuration file and write these values into the file. (Appendix 2.) However, after writing into the file, the “Select Product” spinner’s values are not automatically updated. For updating purposes, a button was created on the main menu that calls a function that updates the spinner’s values by reading the sections in the configuration file.

5 TESTING AND DEPLOYMENT

The testing phase for the application’s development process was conducted Produal’s premises. The testing phase began when the project’s main features were complete. However, parts of the GUI and user feedback features were at this point still unfinished. The program was compiled into an executable file using the auto-py-to-exe converter.

The testing began with setting up one of the commissioner’s computers to run the application. The application required some preparation before it could be used. In order to function properly, the MPLAB X IDE must first be installed, after which the user must change the system’s environment variables. The application requires that the location of the IPECMD executable is added to the Path system variable. This allows the application to activate ipecmd.exe without specifying a separate path in the source code.

After the setup for the application was completed, the testing began with programming a product frequently made by the commissioner so that the application would receive as much testing as possible. During these initial tests, a bug was detected. The bug affected the way the program prints out information on the status of the proramming procedures. Sometimes some of the threads spawned in the run_all function of the application refused to stop running so the message was not printed. This was fixed by forcing the function to cease once either a success message or failure message was found in the output stream. Also, during these initial tests, information was received on the way the application gives the user graphical feedback on what it is doing. The

team leaders suggested that there should be some graphical feedback before the process is finished, so a processing message was added to the application that remains active while the process is underway to better inform the user about the active status. It was also deemed that this better graphical feedback would allow the user to focus on other tasks while the application is running.

In general, during the testing phase, it was also seen that the application reduced the programming time for products to around one third of what it would have been if done manually with a single PICkit-debugger. The testing was successful and no bugs that would break the application were found. One aspect of the application's functionality that was determined worth keeping in mind during deployment and usage of the application was the fact that the computer using the application needs to have access to the paths defined in the configuration file. If the computer does not have access to the paths, the program does not function and will crash.

After the testing was finished, the application was deployed by the production team. The application is currently in use at Produal, and any bugs that may be found in the future are reported to the author as he is currently responsible for the maintenance of the application. During the deployment process, a user manual was written and is presented in Appendix 1.

6 FUTURE IMPROVEMENTS

After the application had been actively used in the deployment phase, it was noted that possible optimization options should be investigated to allow the application to run faster in the future. Due to the use of IPECMD, the application creates log files of all the events occur. Eventually, log files start clogging up the folder where the applications executable is located. A beneficial improvement would be to find a solution to either stop the log files from being created or redirect them to a different folder so as not to disrupt the folder architecture. The configuration file could be moved to a cloud service to remove the need for separate configuration files for different computers.

Another improvement the author has considered is to expand the application to encompass other possible automation solutions and create a single platform for the company's automation needs.

7 CONCLUSION

The purpose of this thesis was to provide insight into the creation and implementation process of the PICkit multiprogrammer application. All the features that the author and the commissioner's representatives agreed upon in their meetings were implemented successfully in the application, and the application has proved to provide a good basis upon which to build new features. The application still has many aspects that could be improved. The GUI of the application should be updated to provide the application with a more professional visual appearance. The configuration file that the application uses could be improved by moving it to a cloud service and having the application access it from there. This would provide the application with uniformity and remove the need to have separate configuration files stored on each individual computer.

Parts of the application could be used to further automate some of the production processes since, for example, the command line interface could likely be used to perform also other functions rather than simply controlling PICkits. The application is currently used by Produal and seems to be functioning well in compliance with all the requirements set by the commissioner.

For the author, the creation of the application provided excellent experience in software production. It provided insight into the importance of making detailed specifications and designs even for a smaller program such as the multiprogrammer. Had the principles of the waterfall model been followed even closer, it would likely have made the implementation stage easier and more time could have been dedicated on other parts of the project. The working process also highlighted the importance of communication with other members of the team when making an application to ensure that all requirements are met. Working on the application also offered various opportunities to learn very useful programming skills and gain knowledge of Python and programming in general.

References

Bell, D. 2005. Software Engineering for Students, Fourth edition. eBook. Available: http://web.firat.edu.tr/mbaykara/softwareengineering.pdf [Accessed 19 April 2020]

Cooling, J. 2019. The Complete Edition - Software Engineering for Real-Time Systems. eBook. Available: www.packtpub.com [Accessed 14 May 2020].

Functional Requirements vs Non-Functional Requirements: Key Differences. No date. Guru99. WWW document. Available: https://www.guru99.com/functional-vs-non-functional-requirements.html [Accessed 4 May 2020].

GSMArena. No date a. microSDHC – definition. WWW document. Available: https://www.gsmarena.com/glossary.php3?term=microsdhc [Accessed 19 May 2020].

GSMArena. No date b. SD (Secure Digital) - definition. WWW document. Available: https://www.gsmarena.com/glossary.php3?term=sd [Accessed 19 May 2020].

glob – Filename pattern matching. No date. PyMOTW. WWW document. Available: https://pymotw.com/2/glob/ [Accessed 4 May 2020].

IEEE-SA Standards Board. 1998. IEEE Recommended Practice for Software Design Descriptions. PDF document. Available: https://www.cs.helsinki.fi/group/linja/resources/IEEE_Std_1016-1998.pdf [Accessed 14 May 2020].

Kent, L. 2014. Python Programming Fundamentals, Second edition. eBook. Available: https://link.springer.com/content/pdf/10.1007%2F978-1-4471-6642-9.pdf [Accessed 18 May 2020]

Lucidchart. No date. What is an Entity Relationship Diagram (ERD)?. WWW document. Available: https://www.lucidchart.com/pages/er-diagrams [Accessed 18 May 2020].

Marsic, I. 2012. Software Engineering. EBook. Available: https://www.ece.rutgers.edu/~marsic/books/SE/book-SE_marsic.pdf [Accessed 19 April 2020].

MPLAB® X IDE User's Guide. 2019. Microchip Technology Inc. PDF-document. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/50002027E.pdf [Accessed 4 May 2020]

MPLAB IPE User's Guide. 2020a. Microchip Technology Inc. PDF-document. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/Integrated%20Programming%20Environment%20User's%20Guide%2050002227E.pdff [Accessed 4 May 2020]

MPLAB® PICkit™ 3 In-Circuit Debugger User's Guide. 2013. Microchip Technology Inc. PDF-document. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/52116A.pdf [Accessed 19 April 2020]

MPLAB® PICkit™ 4 In-Circuit Debugger User's Guide. 2020b. Microchip Technology Inc. PDF-document. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/MPLAB%20PICkit%204%20In-Circuit%20Debugger%20User's%20Guide%2050002751E.pdf [Accessed 19 April 2020]

Parikh, M. 2018. Advantages Of Python Over Other Programming Languages. eLearningindustry. WWW document. Available: https://elearningindustry.com/advantages-of-python-programming-languages [Accessed 14 May 2020]

Parker, J. 2016. Python: An Introduction to Programming. eBook. Available: https://kaakkuri.finna.fi/ [Accessed 14 May 2020]

Phillips, D. 2014. Creating Apps in Kivy. eBook. Available: https://learning.oreilly.com/ [Accessed 4 May 2020].

Produal Ltd. No date. About us. WWW document. Available: https://www.produal.com/about-us/ [Accessed 4 May 2020].

Produal Ltd. No date. Basic Facts. WWW document. Available: https://www.produal.com/about-us/basic-facts/ [Accessed 4 May 2020].

Project Management Docs. No date. Use case document. WWW document. Available: https://www.projectmanagementdocs.com/template/project-documents/use-case-document/#axzz6MP7DA0Lh [Accessed 14 May 2020].

Python range (). No date. Programiz. WWW document. Available: https://www.programiz.com/python-programming/methods/built-in/range [Accessed 4 May 2020].

Sommerville, I. 2016. Software Engineering, 10th edition. eBook. Available: https://dinus.ac.id/repository/docs/ajar/Sommerville-Software-Engineering-10ed.pdf [Accessed 19 April 2020].

String Formatting. No date. learnPython.org. WWW document. Available: https://www.learnpython.org/en/String_Formatting [Accessed 4 May 2020].

subprocess — Subprocess management. No date. Python. WWW document. Available: https://docs.python.org/3/library/subprocess.html [Accessed 4 May 2020].

TechTerms. No date. USB. WWW document. Available: https://techterms.com/definition/usb [Accessed 19 May 2020].

Visual Paradigm. No date. UML Class Diagram Tutorial. WWW document. Available: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/ [Accessed 14 May 2020].

What is Python. No date. Python Software Foundation. WWW document. Available: https://www.python.org/doc/essays/blurb/ [Accessed 19 April 2020].

LIST OF FIGURES

LIST OF LISTINGS