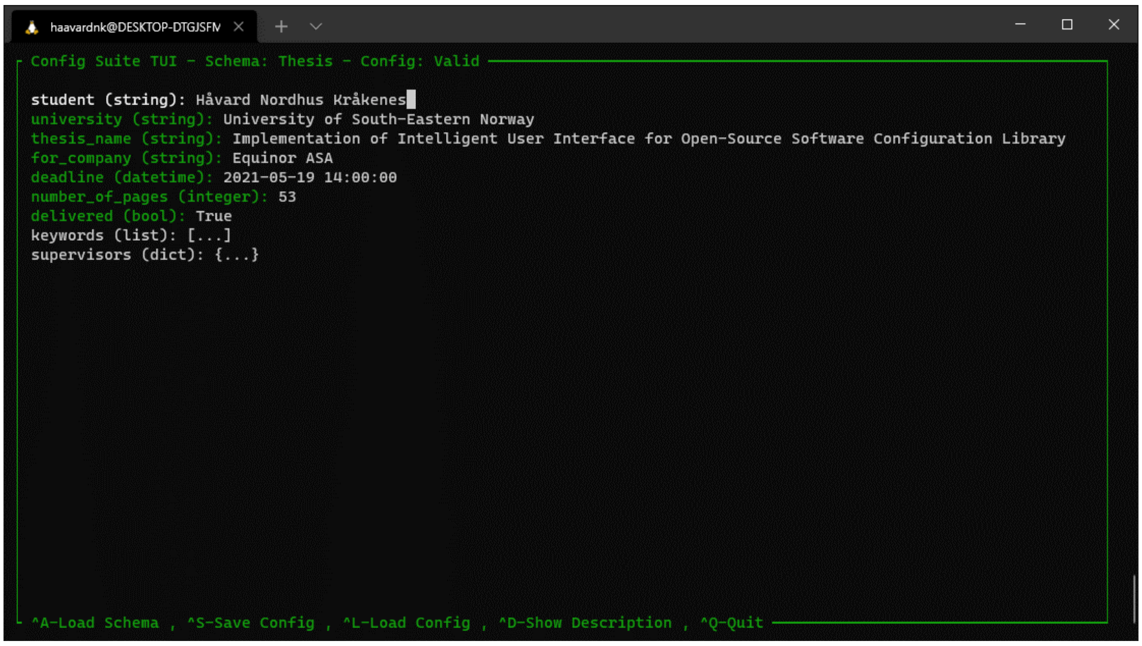
University of
South-Eastern Norway

www.usn.no

FMH606 Master's Thesis 2021

Industrial IT and Automation Online

Implementation of Intelligent User Interface for Open-Source Software Configuration Library



```
Config Suite TUI - Schema: Thesis - Config: Valid

student (string): Håvard Nordhus Kråkenes
university (string): University of South-Eastern Norway
thesis_name (string): Implementation of Intelligent User Interface for Open-Source Software Configuration Library
for_company (string): Equinor ASA
deadline (datetime): 2021-05-19 14:00:00
number_of_pages (integer): 53
delivered (bool): True
keywords (list): [...]
supervisors (dict): {...}

^A-Load Schema , ^S-Save Config , ^L-Load Config , ^D-Show Description , ^Q-Quit
```

Håvard Nordhus Kråkenes

Faculty of Technology, Natural sciences and Maritime Sciences
Campus Porsgrunn

University of South-Eastern Norway

www.usn.no

Course: FMH606 Master's Thesis, 2021

Title: Implementation of Intelligent User Interface for Open-Source Software Configuration Library

Number of pages: 53

Keywords: Software Development, Python, Agile, Software Configuration, Intelligent User Interface, Text-based User Interface, Open-Source

Student: Håvard Nordhus Kråkenes

Supervisor: Hans-Petter Halvorsen

External partner: Equinor ASA

Summary:

Configuration is one of many challenges of software development. Modern software requires configuration in one form or another, often by input parameters stored in configuration files.

Equinor has developed an open-source configuration library in Python called Config Suite for working with configuration files. This library is used to create a schema, a "blueprint," of what the input parameters for the application should look like and parses it on runtime. This means that the developer using this library will know everything about the expected parameters inside the application and will not get invalid parameters.

The application's users have to fill out a configuration file that is correct according to the schema made by the developer. This can be a challenging task, especially for applications with large configurations. This challenge is what this thesis aims to solve by developing an intelligent user interface extension for Config Suite to help the users in filling out a valid configuration according to the given schema.

The extension is developed in Python using Agile methodologies and modern open-source development tools. The result is an efficient Text-based User Interface (TUI) that provides an excellent solution to this thesis's challenge. Being open-source and building on Equinor's open-source first strategy means the software is free to use, change, or extend by anyone in the future.

The University of South-Eastern Norway takes no responsibility for the results and conclusions in this student report.

Preface

This Master's thesis has been written in the spring of 2021 as part of the course FMH606 Master's Thesis at the University of South-Eastern Norway. I have been an online student on the course Master of Science in Industrial IT and Automation for three years while working full time in Equinor.

The project originates from my passion for programming and coding, thus reaching out to our company's software development teams searching for exciting tasks. This project has been composed with Markus Fanebust Dregi, which has also been the external supervisor for this project.

These last three semesters have been both special and challenging because of the Covid-19 pandemic. In this unique situation, I have to thank the university and its teachers for being exceptional in facilitating and helping the students to the best of their ability.

Finally, I would like to thank my supervisors Hans-Petter Halvorsen from USN and Markus Fanebust Dregi from Equinor, for their excellent cooperation in bringing this project forward, and Andrea Brambilla from Equinor for helping organize the project in the early phases.

Bergen, 19.05.2021

Håvard Nordhus Kråkenes

Contents

Abbreviations

Abbreviation	**Definition**
AI	Artificial Intelligence
ML	Machine Learning
CS	Computer Science
IUI	Intelligent User Interface
HCI	Human Machine Interface
NLP	Natural Language Processing
NLU	Natural Language Understanding
UI	User Interface
GUI	Graphical User Interface
TUI	Text-based User Interface
SSH	Secure Shell
OOP	Object-Oriented Programming
TDD	Test-Driven Development
CI	Continuous Integration
OS	Operating System
IP	Intellectual Property
TDD	Test-Driven Development
XML	Extensible Markup Language
YAML	A recursive acronym for “YAML Ain’t Markup Language.”
JSON	JavaScript Object Notation
HTML	HyperText Markup Language

1 Introduction

1.1 Background

Software development and usage of said software can offer many challenges; configuration is one of them. Modern software requires configuration in one form or another, either as the software's settings or as input parameters.

Examples of software settings can be storing the user's choice of the software's display language or window size. Input parameters can be settings and parameters for a physics simulation. These configurations will typically be stored in a file in the software's directory.

The software developer is faced with software configuration challenges all the time. If the input to the software is wrong, the software will fail. In small programs, this will probably just come out as an error message right away, but for complicated software, with extended processing time, it could take hours before reaching the line in the code where it ultimately fails.

For the user, the challenge is the opposite. How do they know what to put in the configuration file? How can they be sure that their settings are valid before launching the software? Should they just try and see if an error occurs? All of this will most certainly result in a bad user experience.

The Software Innovation department in Equinor has faced these challenges and tried to solve them by developing a tool called Config Suite (Equinor ASA, 2018a).

This tool creates a schema, a "blueprint," of what the configuration should look like, both in structure, field names, and accepted values. When the software is started, the software's input is verified either as valid or invalid. It will also give out understandable error messages to the user about what is wrong with specific configuration parameters.

This does solve all the developer's challenges by allowing them to write code without thinking about the possibility of having invalid parameters in their code.

The users will now know that their configuration is valid if the software runs without raising an error on execution. It will even give some understandable feedback on what is wrong if it does not. However, this still requires the user to know what to write in the configuration file in the first place. What if the user only uses the software once every other month? What if the configuration has hundreds or thousands of lines to fill? The user will most certainly not remember what everything in the configuration file means and will often have to read the documentation or seek others' advice.

The objective of this Master's thesis is to solve the users' challenges.

1.2 Objective

This Master's thesis project aims to extend Config Suite (Equinor ASA, 2018a) with an intelligent user interface to help developers and users with the process of filling configuration files according to Config Suite's schema.

Several approaches will be assessed and discussed later in this thesis. The focus is function over form, and it will be developed based on actual user requirements and feedback. The project's goal is to have a tool that many Config Suite users will use in their daily work.

1.3 Methods

Development of the tool will be done in close collaboration with the Software Innovation department in Equinor. The development process will, therefore, replicate how Equinor does software development internally.

1.4 Scope

This project's scope is to develop a user interface extension for Config Suite (Equinor ASA, 2018a). It will be a standalone package and not built into Config Suite.

1.5 Thesis organization

Chapter 2 introduces software configuration and its challenges.

Chapter 3 introduces Config Suite, how it works and what this project aims to extend.

Chapter 4 discusses intelligent user interfaces and its use in this project.

Chapter 5 presents and compares the different concepts evaluated for this project.

Chapter 6 describes the development process and the used tools and resources.

Chapter 7 presents the results of the software development.

Chapter 8 discusses the findings and development of the project.

Chapter 9 presents the conclusion.

This thesis follows the IMRAD model, short for Introduction, Method, Results, and Discussion. The thesis is written for an audience with a technical background and some insight into software development.

2 Software Configuration

2.1 Background

Modern software often supports or requires many external options and parameters to be specified by the user. These are usually stored in one or more configuration file(s). The configuration file can define parameters, options, settings, and preferences for the software.

The configuration files are written with plain text in a serialization format, e.g., Extensible Markup Language (XML), YAML, or JavaScript Object Notation (JSON). These formats are used to represent complex data structures in a way that is easily stored and parsed. Normal file extensions for configuration files are .cnf, .conf, .cfg, and .ini (Bigelow, 2018), though the files can have any extensions. The more descriptive extensions like .json, .xml and .yml is commonly used by developers.

Examples of configuration parameters can be the application's locale settings, input data for processing by the application, or a list of experiment results. An example configuration, which is a primitive product database in JSON format, can be seen in Figure 1.

```
{
    "id": "5968dd23f",
    "product_name": "Ice Cream",
    "supplier": {
        "name": "Ice Cream Co.",
        "address": "Ice Cream Lane 123",
        "active": true
    },
    "quantity":123,
    "unit_cost":14.99,
    "last_restock":"2021-03-28T08:15:30"
},
{
    "id": "56768dh23t",
    "product_name": "Heavenly Coffee",
    "supplier": {
        "name": "Heavenly Coffee Co.",
        "address": "Coffee Lane 321",
        "active": true
    },
    "quantity":321,
    "unit_cost":49.99,
    "last_restock":"2021-03-28T12:25:30"
},
```

Figure 1: Configuration example with JSON formatting

This example will be used in the coming subchapters.

2.2 Challenges

There are many challenges when working with configuration files. Some apparent challenges are parameter naming, parameter type, file formatting, allowed values, or if a parameter is required or optional. This is illustrated in Figure 2.

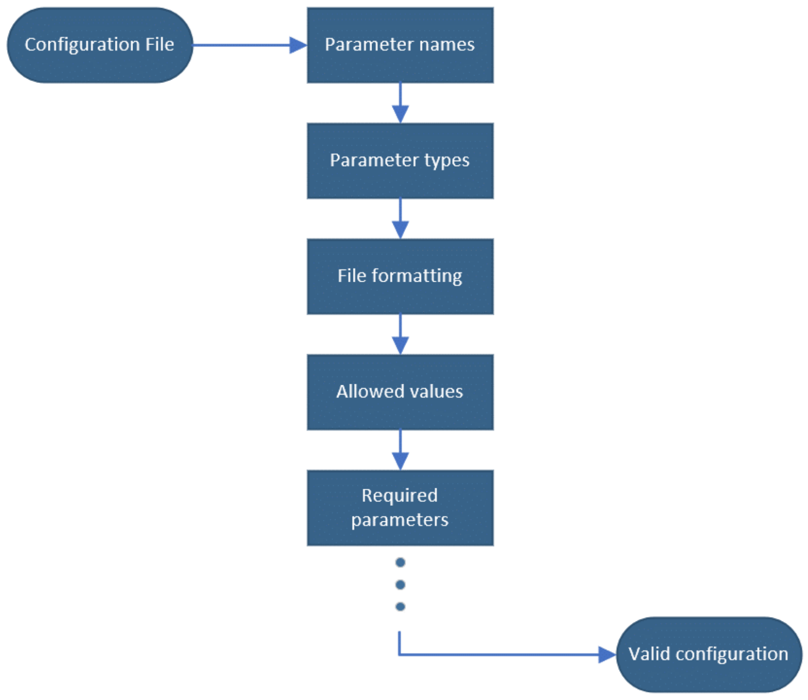


Figure 2: Visualisation of configuration file challenges

The names of the parameters must be precisely the same as what is anticipated by the application. The example in Figure 1 has just nine different names per database entry and will not propose a challenge if the objective is to add another entry to the database. It can, however, propose a challenge if the configuration file is empty. When reading the configuration file programmatically, each parameter's name is essential to connect with the software's corresponding parameter.

In a programming language, all parameters have a defined type. These types are according to a set of rules, usually set by the programming language type settings. In the configuration example in Figure 1, we can see many different parameter types, including String, Boolean, Integer, Float, and DateTime. However, there is no information about what type each parameter should be or how to format the DateTime. The challenge with parameter types is that each parameter written in the code expects to get a specific parameter type. If the parameter type imported from the configuration file is wrong, the software may not load it successfully.

When working with large configurations, it can be challenging to have correct file structure, such as correct indentation and control of the file's different brackets. The example in Figure 1 has two indentation levels, but imagine a configuration file with over 1000 lines and lots of indentations. The complexity of maintaining a configuration file manually will escalate exponentially as its size increases.

Entering parameters in a plain text file is a manual task with no feedback. It will be hard to know if the application requires all parameters or if some of them are optional. There could also be strict rules on which values each parameter can hold. For example, these rules can be exact naming, values in an interval between X and Y, or minimum or maximum length.

The files can come pre-filled with examples, but the documentation will often have to be utilized to make a valid configuration without any background knowledge.

2.3 Data validation and parsing

Validation and parsing are two very similar but slightly different ways of checking if a configuration parameter is equal to its designed blueprint.

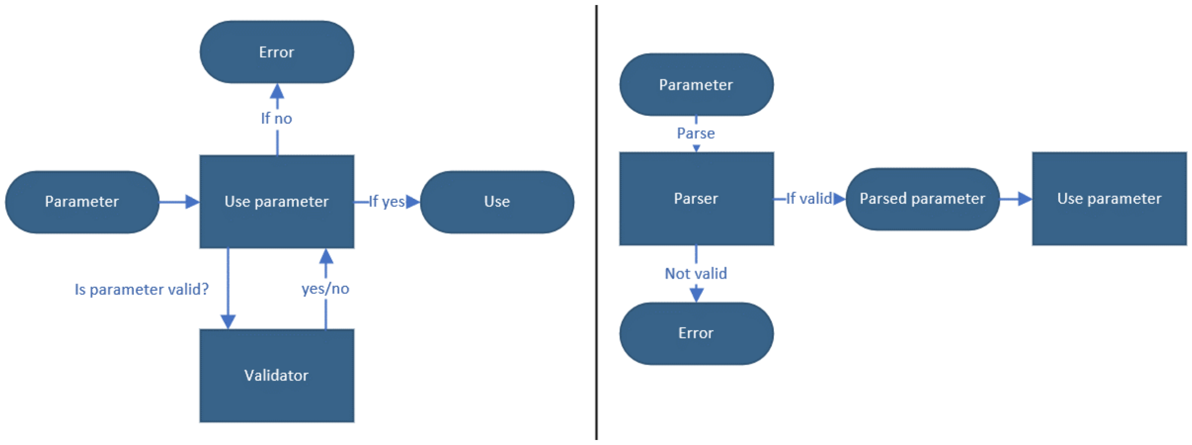


Figure 3: Flow chart comparison of validation vs. parsing

2.3.1 Validation

Validation is a way of checking that the data entered is correct according to the given parameter's designed usage. We usually write a validator function that returns a Boolean for whether the data is valid or not. If the validator returns false, it would also be possible to raise an error to the user.

An example of a validator function written in Python from (Equinor ASA, 2018b) can be seen in Figure 4. This function checks if the passed parameter is a name and returns a Boolean. This is done by checking that the data is a string that only holds alphanumeric characters or spaces.

```python
def is_name(name):
    return all(char.isalpha() or char.isspace() for char in name)
```

Figure 4: Example of a validator function

The way the validator works will only guarantee a valid result at the time of validation. If the validated parameter is changed after previously being validated, it would have to be revalidated. Due to this, validation often has to be done many times throughout the code, as illustrated in Figure 3 above.

Since Python is a strongly typed language (Wikipedia contributors, 2021), meaning the interpreter keeps track of all parameter types, it will also automatically raise a type error if any other types than a string are passed. Python is further discussed in chapter 6.2.

2.3.2 Parsing

Parsing builds on validation, but where a validator only returns a Boolean, a parser will also return the information it learned (King, 2019). In other words, a parser also returns the validated data. This way, the software can ensure that all functions that use the parser's result will work with valid data.

An example where the previous example has been rewritten as a parser can be seen in Figure 5.

```python
def is_name(name):
    if all(char.isalpha() or char.isspace() for char in name):
        return name
    else:
        raise NameError("The parameter is not a name")
```

Figure 5: Example of a parser function

Here, the function returns the name itself if the validation is true. If it is not, it will raise an error.

When comparing validation and parsing, it is a significant benefit to using parsing. Validation is typically done multiple times throughout the code, while parsing allows for doing all checks at the start of the program execution. After the data has been parsed, it does not have to be rechecked to know if it is valid or not (King, 2019). This is illustrated in Figure 3 above. Config Suite is a Python package that takes advantage of the benefits of parsing.

3 Config Suite

As stated in the introduction, Config Suite is Equinor's self-developed tool for configuring software. This section will explain what Config Suite is and what it tries to accomplish, focusing on the parts applicable to this project. This chapter's information is collected from Config Suite's documentation (Equinor ASA, 2018b).

3.1 Challenges it solves

Config Suite is designed to parse external software configurations in YAML formatting. The configurations are typically passed to Config Suite at the start of the program execution. After Config Suite parses the configuration, it creates an immutable configuration object, meaning that it is no longer possible to change it without re-parsing. This is a significant advantage, as configuration software with mutable configurations also tends to inject software state into the configuration, which is generally an anti-pattern. With an immutable configuration, the developer has complete control over the available data in the code and does not have to worry about invalid data.

Config Suite also provides its users with meaningful feedback if there are errors in their configuration.

A figure showing the use of Config Suite can be seen in Figure 6.

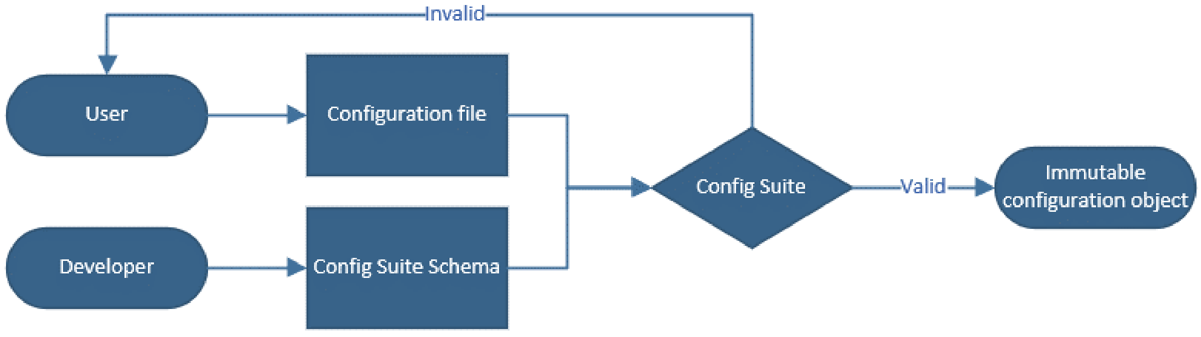


Figure 6: A basic flow chart showing the use of Config Suite

3.2 How it works

This part will focus on how Config Suite works by going over some of the basic concepts of Config Suite.

3.2.1 Schema

The core functionality in Config Suite revolves around what is called a schema. The schema is a blueprint of all expected parameters of the host software and is defined by the developer. An illustration showing the contents of the schema can be seen in Figure 7.

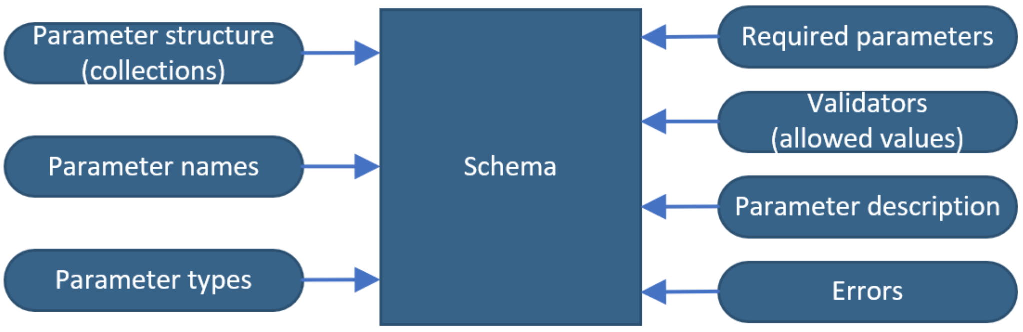


Figure 7: Visualisation of inputs to a schema

Basic schemas define expected parameter structures, names, and types. More advanced schemas can define custom parameter types, required parameters, validators, description messages, and error messages. An example of a basic schema can be seen in Figure 8.

```
from configsuite import types
from configsuite import MetaKeys as MK

schema = {
    MK.Type: types.NamedDict,
    MK.Content: {
        "name": {MK.Type: types.String, MK.Description: "Name of person"},
        "hobby": {MK.Type: types.String, MK.Description: "Persons hobby"},
        "age": {MK.Type: types.Integer, MK.Description: "Age of person"},
    }
}
```

Figure 8: Example of a basic Config Suite schema

To use the software, the user has to make a valid configuration based on the schema. The configuration will only be valid if all the rules and requirements of the schema are fulfilled. If the configuration is valid, an immutable Config Suite configuration object is made, which in turn will be used by the software for all configuration parameters.

Config Suite has built-in support for a set of basic parameter types. The basic types are String, Integer, Number, Boolean, Date, and DateTime. These types are all supplied with standard type validators and errors. The developer can create custom parameter types with custom validators and errors to use in their schema. The basic parameter types will be supported by the UI developed in this project. Custom parameter types can easily be added at a later stage.

3.2.2 Collections

The supported collections in Config Suite are List, Dict, and Named Dict. The List is a sequence of parameters of arbitrary length, Dict is a dictionary with arbitrary keys, and Named Dict is a dictionary where all the keys are known upfront. Lists and Dicts can only hold one type of parameter, while the Named Dict can have parameters of different types. Except for only holding specific types, they all work the same as Python Lists and Dictionaries, which can be read more about in chapter 6.2.1.

The UI will support all collections from Config Suite.

3.2.3 Error messages

If there are errors when validating a configuration, Config Suite will raise a list of errors to the user. It will inform the user of type errors, naming errors, and structural errors out of the box. The developer can also implement custom error messages and type errors. An example error output when trying to enter a String into the age field of the example in Figure 8 can be seen in Figure 9 below.

```
(InvalidTypeError(msg=Is x an integer is false on input '45', key_path=('age',), layer=None),)
```

Figure 9: Example of a Config Suite error

These error messages will be extracted and used in the UI to show the user where there are errors in the configuration.

3.2.4 Readable

If a configuration is not valid, there is another validation function in Config Suite called readable. This function checks whether the configuration follows the correct form of the schema so that it can be used to fill in the schema even if the values are not valid.

An example of this is entering a String in an Integer field, it is not valid, but the software can still read it.

3.2.5 Automatic generation of documentation

Config Suite has a built-in function to generate documentation for a given schema. This can be done either programmatically or as a Sphinx extension. More details on Sphinx can be seen in section 6.4.5.

Programmatic generation is done by passing the schema to the command `configsuite.docs.generate(schema)`. This returns documentation for the schema as reStructuredText (Goodger, 2021). The generated documentation for the example schema in Figure 8 can be seen in Figure 10 below.

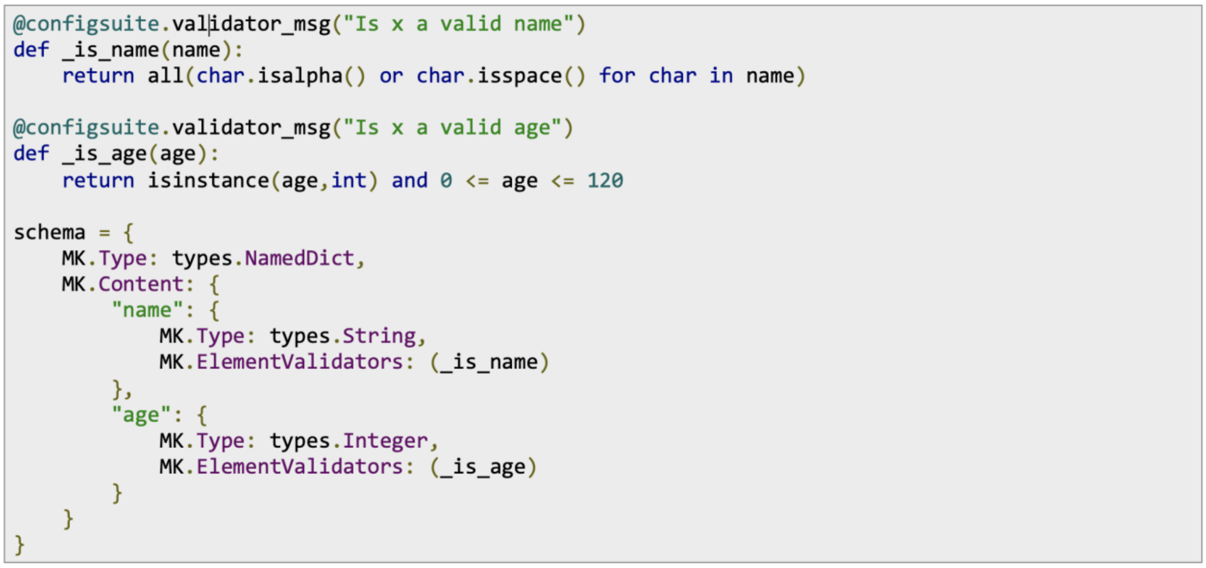
```
name*:
    Name of person
    type: string
hobby*:
    Hobby of person
    type: string
age*:
    Age of person
    type: integer
```

Figure 10: Example of Config Suite generated documentation

3.2.6 Validators

Config Suite allows the developers to write custom validators, making it possible to validate parameters in other ways than just type checking. Examples of this can be to validate that a name only contains alphabetic characters and spaces or that the age is between 0 and 120.

An example of a schema using these two validators can be seen in Figure 11 below.



```python
@configsuite.validator_msg("Is x a valid name")
def _is_name(name):
    return all(char.isalpha() or char.isspace() for char in name)

@configsuite.validator_msg("Is x a valid age")
def _is_age(age):
    return isinstance(age,int) and 0 <= age <= 120

schema = {
    MK.Type: types.NamedDict,
    MK.Content: {
        "name": {
            MK.Type: types.String,
            MK.ElementValidators: (_is_name)
        },
        "age": {
            MK.Type: types.Integer,
            MK.ElementValidators: (_is_age)
        }
    }
}
```

Figure 11: Example of a Config Suite schema using validators

3.3 Current user workflow

All use of Config Suite is currently done by code, either in a Python shell or in a Python script. Usually, the schema and schema usage are hardcoded as part of a Python package, where it is up to the user of the package to fill in a valid configuration file to use it. This can be a challenging task depending on the supplied documentation and the availability of pre-filled configuration files. Some configurations can even be over 1000 lines long. This makes it hard to know what to write in each configuration file line, especially if one rarely uses the Python package in question. A figure showing the current user workflow is shown above in Figure 6.

This is one of the challenges this project is trying to solve by creating an intelligent user interface intended to help users fill out schemas in configuration files.

4 Intelligent user interfaces

This chapter will in detail go through and reflect on the creation of an Intelligent User Interface (IUI) for the Config Suite software.

4.1 What is intelligence in user interfaces?

The word “intelligent” in a user interface can be interpreted in many different ways. The most common interpretation is the intersection between Human-Computer Interaction (HCI) and Artificial Intelligence (AI) (Völkel et al., 2020). In academia and research, the term IUI is used for a user interface with intelligent features.

In a paper by (Völkel et al., 2020), they analyzed 25 years of research papers to determine what terms define an intelligent user interface. Though the terms have gotten more diverse as time passes, the most used terms for describing the IUI itself were; adaptation, automation, and interaction. An IUI also performs actions, in which the terms they found to describe the actions done by an IUI were; assist, create, detect, and adapt.

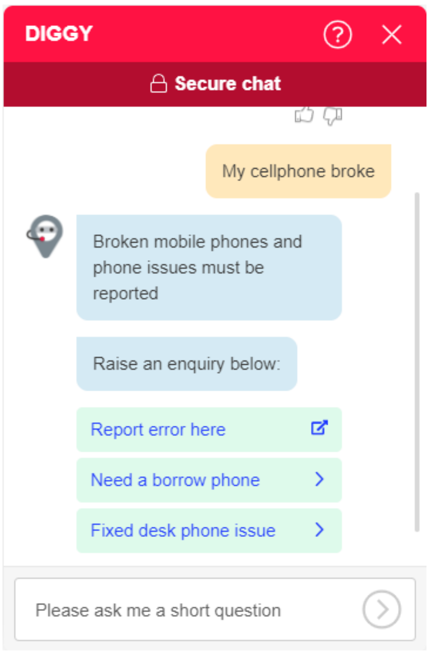
These are just some of the many terms that can describe intelligent user interfaces and highlight that the word “intelligent” is a broad expression.

From this abstract definition, we will explore some systems commonly characterized as “intelligent.”

4.2 Examples of intelligent user interfaces

The most common IUI of modern times are chatbots, which businesses have widely adopted for messaging apps, customer service, internal services, and much more. These chatbots save countless hours yearly by answering simple questions and automating simple tasks so that humans can focus on the more advanced problems. Many companies even try to hide their human operators behind chatbot walls to save them from the most straightforward questions.

Equinor has a chatbot named Diggy, which is made to help employees find help and information without having to talk to a human. A short chat with Diggy can be seen in Figure 12.


DIGGY

Secure chat

My cellphone broke

Broken mobile phones and phone issues must be reported

Raise an enquiry below:

Report error here

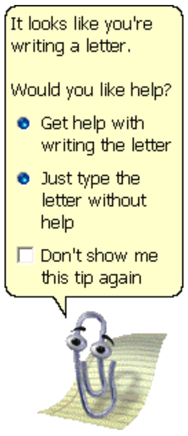
Need a borrow phone

Fixed desk phone issue

Please ask me a short question

Figure 12: Conversation with Diggy, Equinor's chatbot (Equinor ASA, 2021b)

The chatbots can relate to most terms found by (Völkel et al., 2020) as it is an assistant that detects and adapts to the users' questions and interacts with them like a human being. Modern chatbots use many AI and Machine Learning (ML) concepts, such as Natural Language Processing (NLP) and Natural Language Understanding (NLU), Pattern Matching, as well as databases of scripted rules and replies (Adamopoulou & Moussiades, 2020).


It looks like you're writing a letter.

Would you like help?

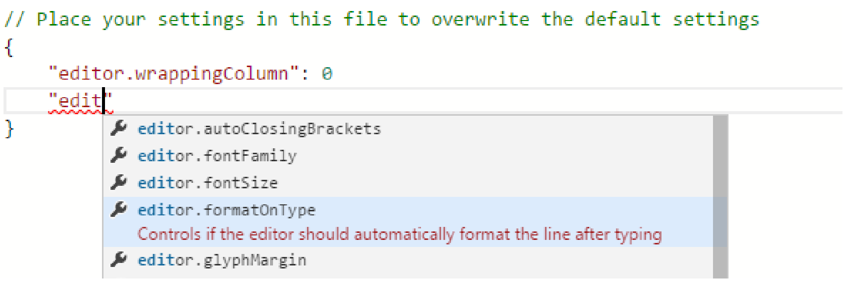
- Get help with writing the letter
- Just type the letter without help

☐ Don't show me this tip again

Figure 13: Microsoft Office Assistant, Clippit, Nickname Clippy (Microsoft, 1997)

Another well-known IUI is Microsoft's office assistant, often nicknamed Clippy, which gave helpful hints in using their Office software suite from 1997-2003 (Microsoft, 1997). Though it was very advanced at the time, it was not very "intelligent," which resulted in people feeling Clippy was intrusive and annoying, and it was later even made fun of by Microsoft themselves.

Though Intelligent User Interfaces can seem "alive" and therefore "intelligent," there are also backend systems that help users complete tasks by automating and adapting to their input. One such system, which also is one of the inspiration sources for this project, is the built-in JSON schema editor in Visual Studio Code.



```
// Place your settings in this file to overwrite the default settings
{
    "editor.wrappingColumn": 0
    "edit"
}
```

```
editor.autoClosingBrackets
editor.fontFamily
editor.fontSize
editor.formatOnType
Controls if the editor should automatically format the line after typing
editor.glyphMargin
```

Figure 14: Visual Studio Code JSON editor (Microsoft, 2021)

This plugin helps the user fill in a JSON schema with auto-completion and can auto validate the input against a user-specified schema. A screenshot from the auto-completion can be seen in Figure 14.

4.3 Interpretation of Intelligence

For this project, the interpretation of intelligence is very similar to that of the JSON editor described in chapter 4.2. The goal is to have a UI that helps the user fill in the Config Suite YAML schemas.

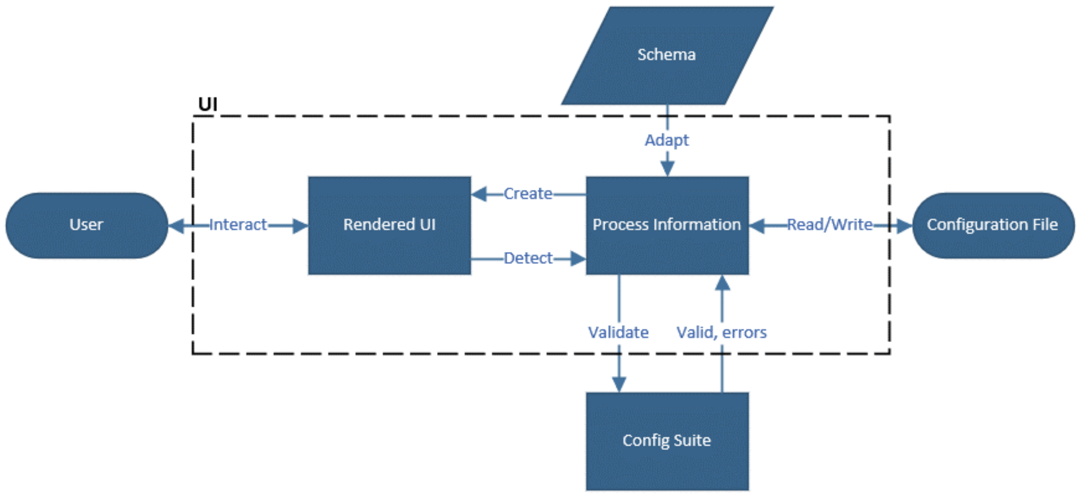


Figure 15: Flow diagram of UI Intelligence

Adaption is, therefore, a vital term since the UI must adapt to the supplied schema. It also has to detect the user's input, interact with the user, and automate as many tasks as possible. A flow diagram highlighting the intelligent functionality wanted from the UI can be seen in Figure 15.

5 Concept selection

This project started as just an idea. This idea was further developed in the early phases of the project, resulting in four different concepts. This chapter will explore the concepts and discuss the final selection.

5.1 Alternatives

The idea from the start was to make an Intelligent User Interface for Config Suite. However, what kind of user interface or what functionality it should have was an open topic. The idea was to make an easy-to-use interface that would help the user fill out configuration files for Config Suite.

During this process, it was vital to keep in mind that the primary users of Config Suite are engineers, developers, and other skilled users that all are proficient in scripting and using Linux as their Operating System (OS). This information let us make many assumptions and choices that would usually be out of the question when creating software. For example, since everyone is proficient in scripting and using Linux, we can assume that they also have command line experience.

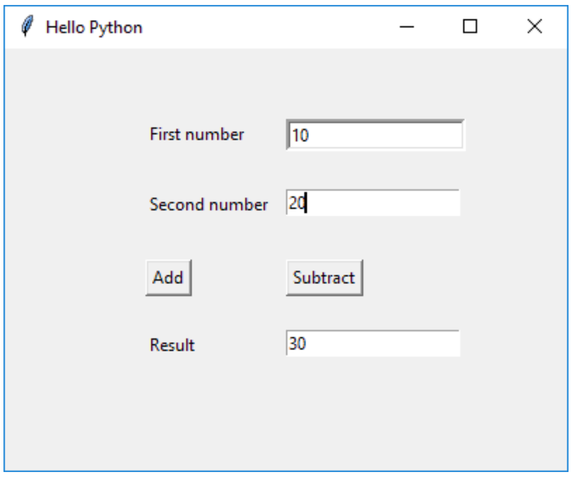
The resulting concepts were:

1. A Graphical User Interface (GUI)
2. A Text-Based User Interface (TUI)
3. An extension for Visual Studio Code
4. An API for Config Suite combined with a UI in a non-python framework

5.1.1 Graphical User Interface

The first concept is a GUI, which is the most common type of UI used in, for example, computers and phones. It is a modern-looking UI with many visual elements and gives a great user experience. In many cases, this would be the obvious choice for a UI in 2021. Python can make modern GUIs for all OSs', but it is not widely adopted since Python focuses more on AI, Computer Science (CS), and other fields that mostly use text or other generated output. This means that the tools to make GUIs in Python could be limited compared to other GUI-capable programming languages like, for example, C#, Java, and JavaScript.

However, even if Python GUIs is unusual and has fewer tools than other programming languages, it is still a viable option. One of the available libraries for making GUIs in Python is Tkinter (Python Software Foundation, 2021b). A screenshot of a GUI made with Tkinter can be seen in figure Figure 16 below.


Hello Python

First number 10

Second number 20

Add Subtract

Result 30

Figure 16: Example of a Python GUI made in Tkinter (TutorialsTeacher, 2020)

5.1.2 Text-Based User Interface

Since the target group in Equinor is a strong adopter of Linux and Python, it made sense to evaluate an option where the user interface is implemented in their most used tool; the terminal.

The second concept is a TUI, a UI that is rendered into the terminal and usually navigated with the keyboard. It has a very classical look to it, which looks like it could have been made in the ’80s. This gives a basic and straight-to-the-point application without any extra fuss. This implementation would be written in Python as a direct extension to Config Suite. An example of a TUI can be seen in Figure 17.

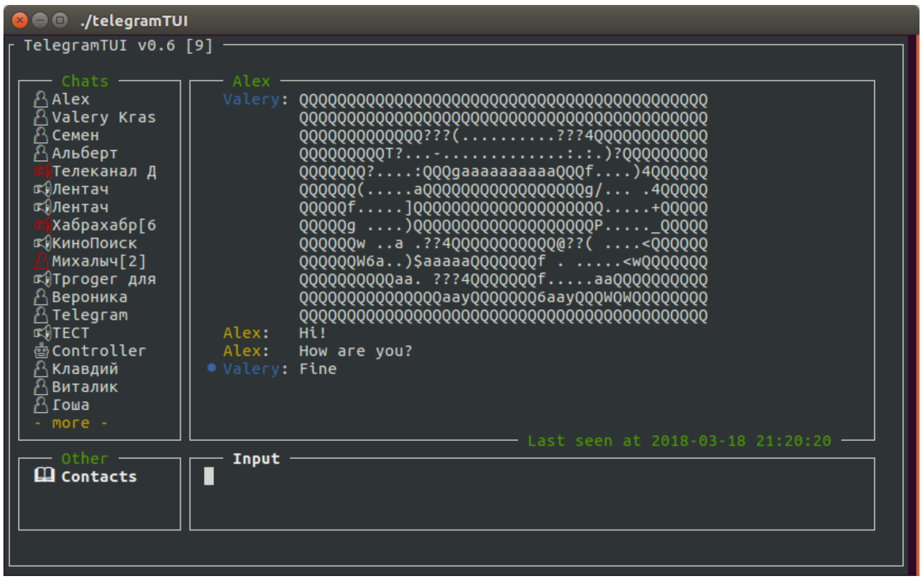

```
./telegramTUI
TelegramTUI v0.6 [9]

 Chats            Alex
 Alex             Valery: QQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQ
 Valery Kras              QQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQ
 Семен                    QQQQQQQQQQQQQ???(..........???4QQQQQQQQQQQQ
 Альберт                  QQQQQQQQQT?...-.............:.:.)?QQQQQQQQQ
 Телеканал Д              QQQQQQQ?....:QQQgaaaaaaaaaaQQQf....)4QQQQQQ
 Лентач                   QQQQQQ(.....aQQQQQQQQQQQQQQQQQg/... .4QQQQQ
 Лентач                   QQQQQf.....]QQQQQQQQQQQQQQQQQQQQ.....+QQQQQ
 Хабрахабр[6              QQQQQg ....)QQQQQQQQQQQQQQQQQQQP....._QQQQQ
 КиноПоиск                QQQQQQw ..a .??4QQQQQQQQQQQ@??( ....<QQQQQQ
 Михалыч[2]               QQQQQQW6a..)$aaaaaQQQQQQQf . .....<wQQQQQQQ
 Трgoger для              QQQQQQQQQQaa. ???4QQQQQQQf.....aaQQQQQQQQQQ
 Вероника                 QQQQQQQQQQQQQQQQaayQQQQQQQ6aayQQQWQWQQQQQQQQ
 Telegram                 QQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQ
 ТЕСТ             Alex:   Hi!
 Controller       Alex:   How are you?
 Клавдий        ● Valery: Fine
 Виталик
 Гоша
 - more -
                                    Last seen at 2018-03-18 21:20:20
 Other            Input
 Contacts         █
```

Figure 17: Example of TUI, TelegramTUI (Krasnoselsky, 2018)

A benefit of this concept is that Python, in general, is very text-based. Most users run applications, scripts, or the Python Shell from their terminal. This solution would let the users run the application from the Python Shell and use the returned data for further processing.

TUIs are also great for remote usage. When configuring software for a Linux system, it does not necessarily have to be on a personal computer. It can very well be for configuring a simulation running on a remote Linux-based server or supercomputer. In Equinor, Linux and Python are widely used for AI, ML, and CS, and many of these tasks are executed on high-performance compute clusters. The TUI could then be launched in a Secure Shell (SSH) remote session.

5.1.3 Visual Studio Code Extension

The third concept is a deviation from the traditional user interface. The concept is to make an extension for Visual Studio Code. Visual Studio Code is a popular text editor made by Microsoft, which has a rich extension system.

The concept here is much looser, as the extension system for Visual Studio Code is quite extensive and open. This could result in, for example, an auto-complete configuration editor inspired by the JSON editor shown in Figure 14 in chapter 4.2 above, a schema loader and filler, or something completely different. This means that this concept needs more work to mature into a final concept.

Another potential problem with this concept is that the extensions for Visual Studio Code are written in TypeScript, which would make it tricky to communicate with Config Suite directly since Config Suite is written purely in Python.

5.1.4 Config Suite API

A fourth concept was proposed and discussed loosely, namely an API wrapper for Config Suite, to easily communicate with other programming languages. This would make it much easier to extend Config Suite with a web GUI, a UI made in another programming language, or make the Visual Studio Code extension more viable.

Therefore, this concept would be considered an enabler, combined with one of the other proposed concepts.

5.2 Discussing the alternatives

The usability of each of the concepts is considered to be roughly equal. This means that the technical differences will be used for evaluating the alternatives.

The Visual Studio Code extension concept has some significant uncertainties: the idea is very open, and that the communication between TypeScript and Python could be a huge problem. This could be solved by combining it with the API, but this option was discarded because of limited time to develop this concept further.

The two other concepts, namely the TUI and the GUI, are much more similar and will be compared in greater detail.

When evaluating the functionality, both will achieve precisely the same, just with a different look and way of controlling the UI. The backend working of the different UIs' could, in theory, be identical.

The GUI provides a much more streamlined and modern look & feel than a TUI. It can also be said to be much more self-explanatory and user-friendly to inexperienced users, according to a study by (Chen & Zhang, 2007). The GUI will have a slight overhead of resource usage and will generally be slower than a TUI. It can also be argued that a nice-looking graphical interface often is designed to be pleasing to the eye, which could result in a form over function design.

The TUI is a very minimalistic UI and is very much function over form because of its design. However, this can often be an advantage when looking to increase a simple task's efficiency.

The study by (Chen & Zhang, 2007) also proved this by testing how long time expert and novice users needed to do the same task in a TUI and a GUI. The novices performed better on the GUI, but the expert users were faster on the TUI. Our users can be considered experts, and therefore this will advantage the TUI.

Another advantage in favor of the TUI is that it is much easier to use remotely. Since a TUI can be launched in an SSH session with a remote compute cluster, it will be much easier to use remotely than a GUI, which requires a screen output from the connected computer. This weights the decision greatly because of the targeted users of the application.

5.3 Concept selection

After assessing the different concepts and discussing with the customer, making a TUI was selected. The reason for choosing this was a combination of the previous subchapter's arguments and that the customer was more intrigued by that concept. In the end, it is all about creating a product that the customer finds useful and enjoys using.

The following chapters will elaborate on the process and tools used to create the TUI and explore the final product.

6 Software Development Background

This chapter will explain the methodologies and tools used for the software development process of this project.

6.1 Agile project management

Agile is a buzzword in project management that has gotten a broad meaning, including methodologies, tools, and processes. However, originally, it is a mindset. It is the result of the Agile Manifesto (Beck et al., 2001) authored by 17 software developers in 2001. The Agile Manifesto is a set of four values and twelve principles.

Today, the ecosystem around Agile is adopted widely in modern software development and project management.

The Agile ecosystem is extensive and is mainly suited for teams and teamwork. Since this Master's thesis is a solo project, only the Agile mindset and a few Agile ecosystem parts were adopted.

6.1.1 User Stories and Epics

User stories are one of the core components of Agile software development. It is a more user-oriented way of describing how computer software should work compared to standard requirements.

User stories build on the Agile mindset of putting people first. They are short stories written by the system's end-user to describe a feature they want in the final product. They usually follow a standard pattern: "As a (type of user), I want (some goal) so that (some reason) (Atlassian, 2021).

Epics are also user stories but are used for the project's more significant milestones and often hold a larger work body. An Epic can be built up of several smaller user stories. Each development iteration, which spans over one to two weeks, usually holds one epic and several user stories.

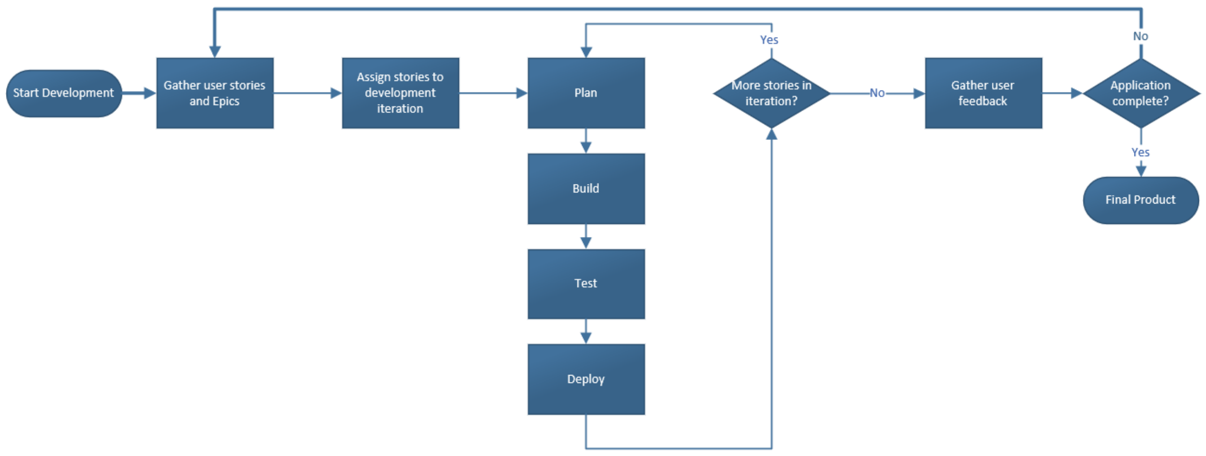


Figure 18: Agile development process

In Agile software development, the development of the software is split up into many smaller parts. The user story represents one part of the system. One complete development cycle (planning, building, testing, deploying) is carried out for each story. Each cycle builds upon the previously completed software by adding another user story’s functionality. This is called incremental development.

Since there is a functioning system after each cycle, it is possible to gather honest user feedback many times during the system’s development. This is usually done between development iterations.

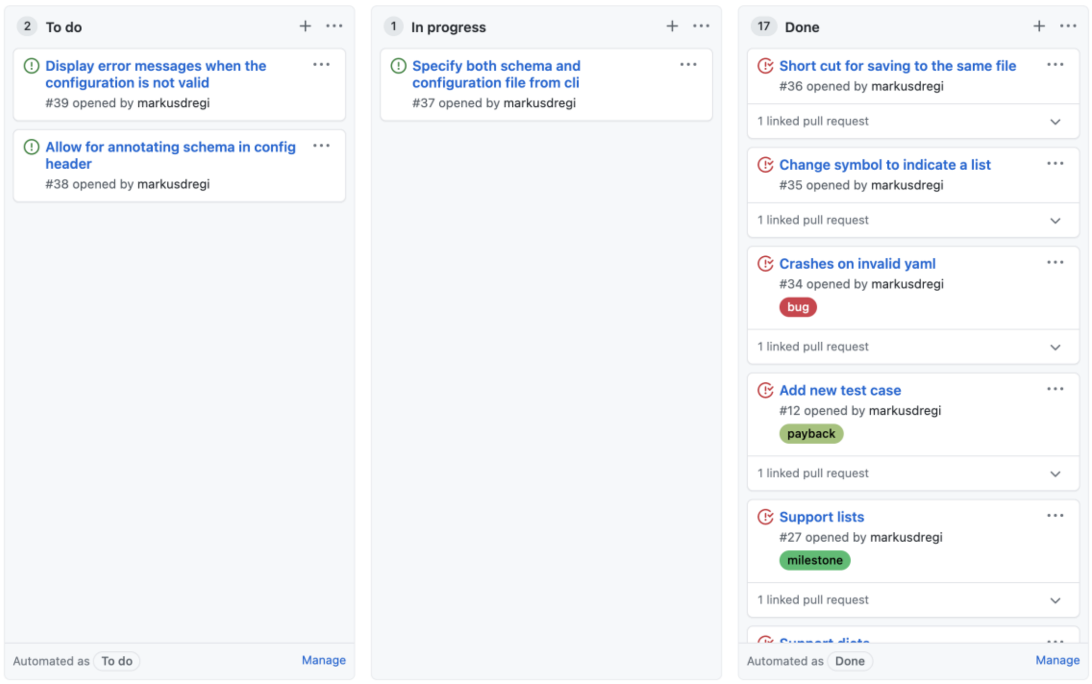
User stories are the foundation of the software development in this project, where the user stories and user feedback are supplied by the customer (Equinor).

A figure of the Agile process used in this project can be seen in Figure 18.

6.1.2 Kanban

A Kanban board is used to keep track of the tasks during the project’s software development. It is a project management tool that is used to visualize work in progress. It is originally a Lean practice that arose as a scheduling system for lean manufacturing at Toyota in the 1940s (Kanbanize, 2021). Since Lean shares many similarities with Agile, it has also been adopted into the Agile ecosystem (Agile Thought, 2021).

All the user stories start on the board’s left side, in the “story queue” or backlog. The user stories move to the right over the board until it reaches the last stage and is complete. The stages the stories go through in this project are; To do, In progress, and Done, where Done means the changes have been deployed.


2 To do

Display error messages when the configuration is not valid
#39 opened by markusdregi

Allow for annotating schema in config header
#38 opened by markusdregi

Automated as To do — Manage

1 In progress

Specify both schema and configuration file from cli
#37 opened by markusdregi

17 Done

Short cut for saving to the same file
#36 opened by markusdregi
1 linked pull request

Change symbol to indicate a list
#35 opened by markusdregi
1 linked pull request

Crashes on invalid yaml
#34 opened by markusdregi
bug
1 linked pull request

Add new test case
#12 opened by markusdregi
payback
1 linked pull request

Support lists
#27 opened by markusdregi
milestone
1 linked pull request

Automated as Done — Manage

Figure 19: Example Kanban board state from this project

On the Kanban board, the number of stories on each stage is limited on purpose, as proposed by (Patton, 2020). A maximum limit of two user stories can be in the “In progress” stage simultaneously. The Kanban board is used for the development part of the project. An example of a Kanban board state from this project can be seen in Figure 19. The entire Kanban board can be seen on GitHub (Equinor ASA, 2021a)

6.1.3 Gantt

Gantt is an old method of making a complete plan for a project and not directly related to Agile. It is related to the waterfall software development method, which is a predecessor of Agile. The Gantt chart is a timeline chart that shows all planned activities and their planned period and is commonly used for higher-level planning. However, it is not a suitable tool for planning software development because it tends to render useless in projects with high uncertainty. Because of this, the Gantt chart is used for planning and keeping track of project milestones and overall tasks, and the Kanban board is used for detail planning of the software development. A screenshot of the Gantt chart used in this project can be seen in Figure 20.

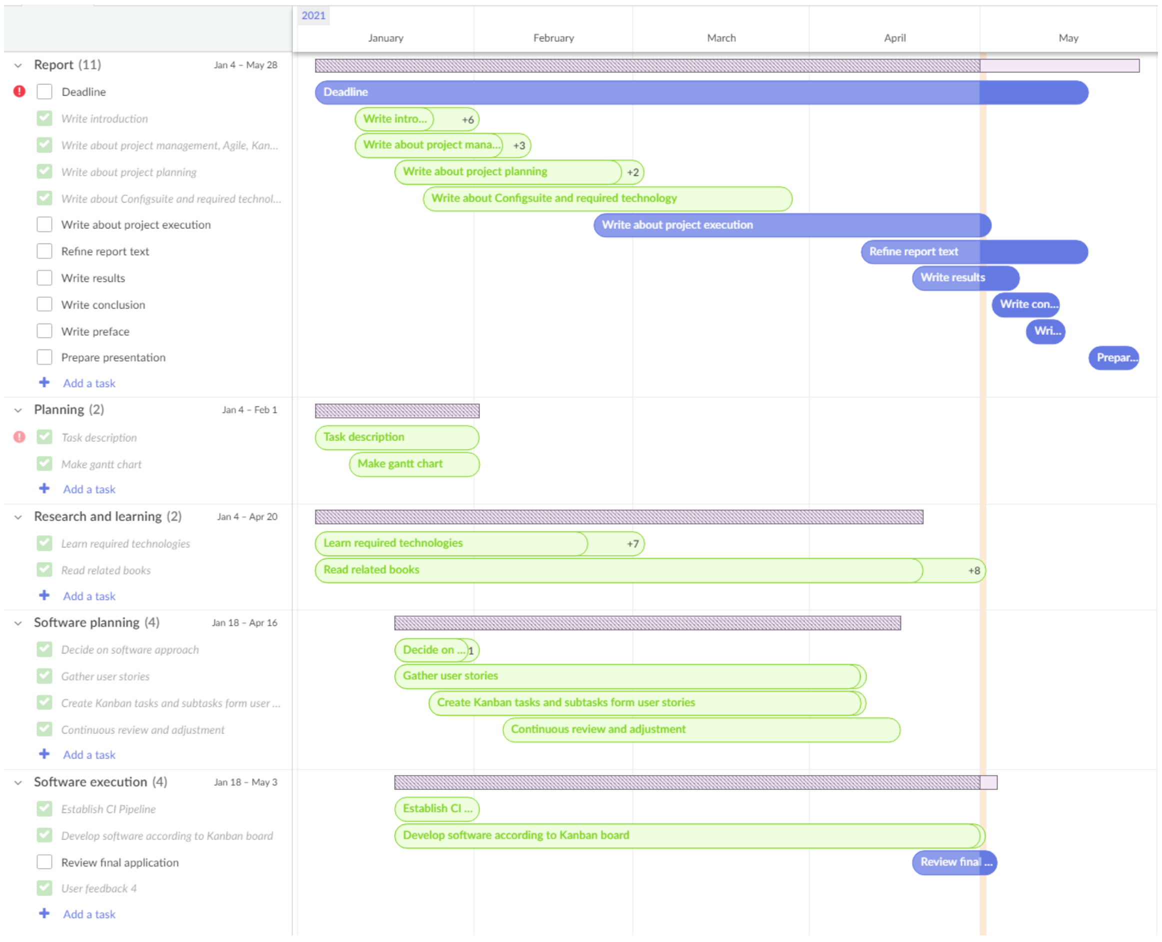


Figure 20: Gantt chart from project planning

6.2 Python language background

Python is the programming language used for this project. Python is a modern and trendy programming language that has risen significantly on the popularity lists in recent years. It was, in fact, the most used programming language in 2020 (Eastwood, 2020). Most of this is due to the rise of AI, ML, and CS.

Even though Python has risen the ranks in later years, it is, in fact, an old programming language. It was first released in 1991 by Guido Van Rossum as a successor to the programming language named ABC (Wikipedia contributors, 2021).

Python is a strongly typed language (Wikipedia contributors, 2021) that comes with much built-in functionality. Furthermore, what is not included can often be found as community-made packages.

6.2.1 Python quick introduction

This subchapter will give a quick introduction to Python's basic concepts central to the development of the TUI.

Parameters: Python uses Dynamic typing, which means that we do not have to declare parameter types implicitly like in, for example, Java (Bailey, 2020). Some of the main built-in parameter types are the String, List, and Dictionary.

A string is an array of bytes representing Unicode characters. Each character can be extracted using its index in the array. In Python, indexes start at zero, whereas many other programming languages start at one.

Lists can store a collection of data, where each element in the List can be of any type. Elements from a list can be extracted using the index of the wanted element, and the List can also be iterated over to either extract info from all elements or look for a specific element based on other parameters.

A Dictionary is similar to a List, except elements are referenced using a string key instead of an index number. A value is appointed to each key in the Dictionary, where the value can be of any type.

Examples of these parameters can be seen in Figure 21 below.

```
# String
this_is_a_string = "This is the string value"

# List
this_is_a_list = [element1, element2, element3]

# Dictionary
this_is_a_dictionary = {key1:value1, key2:value2, key3:value3}
```

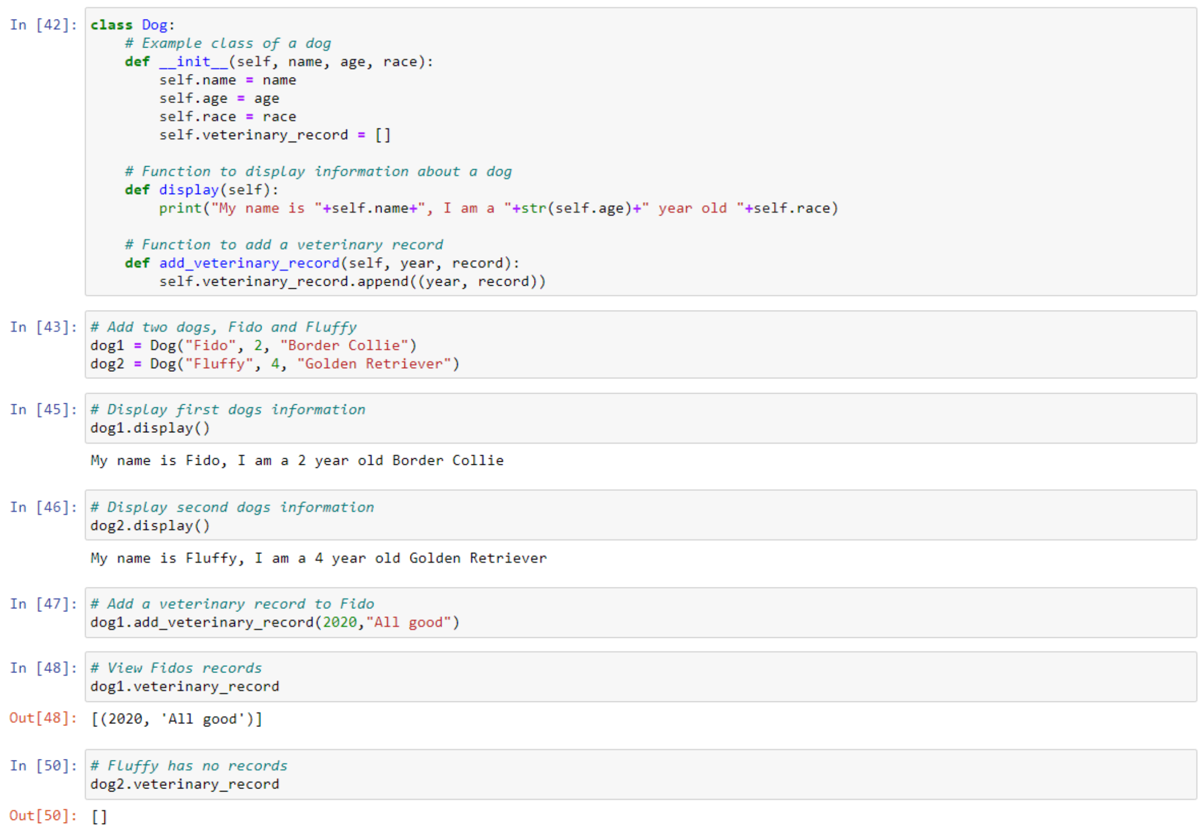
Figure 21: Example of Python parameter types

Indentation: Python does not use a semicolon to indicate indentation or braces to indicate the content of classes and functions like other programming languages like C# and Java does. Instead, it uses line indentation to indicate blocks of code. The number of spaces can vary, but all within the same block must have the same indentation. Examples of indentation can be seen in Figure 22 below.

Classes and functions: Python are an Object-Oriented programming language. Object-Oriented Programming (OOP) is a programming design method where programs are structured to bundle properties and behaviors into classes.

A class represents a new object type, which could, for example, be a dog. The dog class's attributes could be the dog's name, age, race, and veterinary records. The class could also have methods for adding a veterinary record or displaying its information.

Each dog is a new instance of the class, where each instance maintains the state of its attributes and is separate from all the other dogs. An example can be seen in Figure 22.

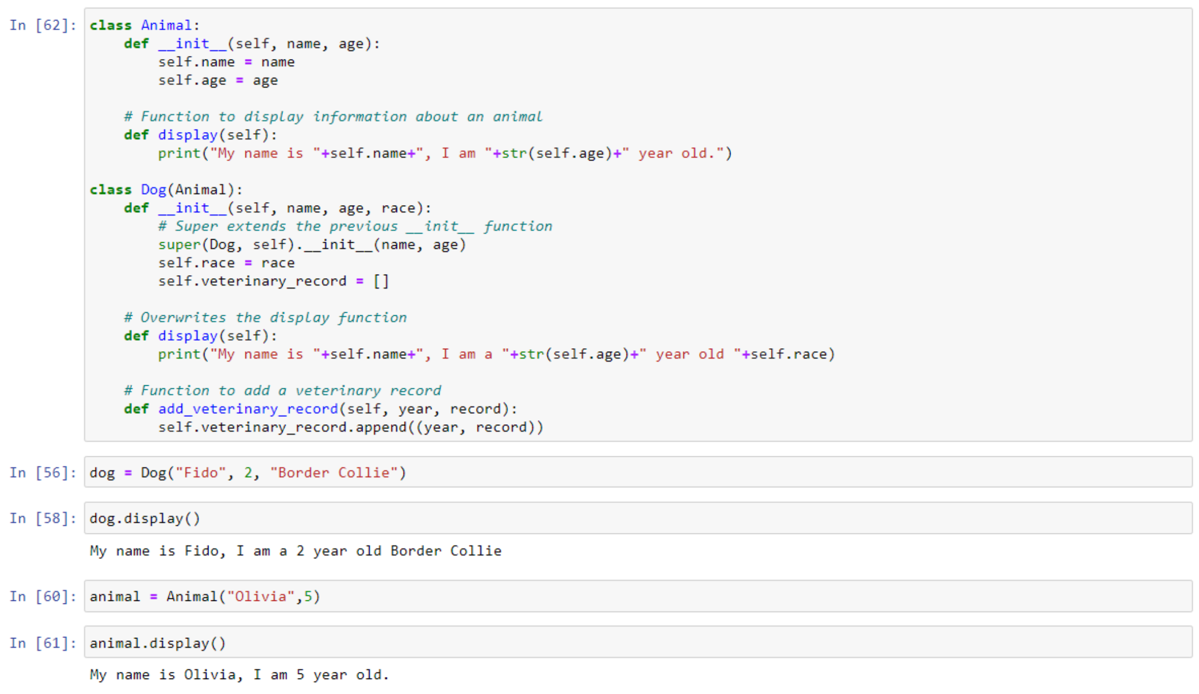


```
In [42]: class Dog:
             # Example class of a dog
             def __init__(self, name, age, race):
                 self.name = name
                 self.age = age
                 self.race = race
                 self.veterinary_record = []

             # Function to display information about a dog
             def display(self):
                 print("My name is "+self.name+", I am a "+str(self.age)+" year old "+self.race)

             # Function to add a veterinary record
             def add_veterinary_record(self, year, record):
                 self.veterinary_record.append((year, record))

In [43]: # Add two dogs, Fido and Fluffy
         dog1 = Dog("Fido", 2, "Border Collie")
         dog2 = Dog("Fluffy", 4, "Golden Retriever")

In [45]: # Display first dogs information
         dog1.display()

         My name is Fido, I am a 2 year old Border Collie

In [46]: # Display second dogs information
         dog2.display()

         My name is Fluffy, I am a 4 year old Golden Retriever

In [47]: # Add a veterinary record to Fido
         dog1.add_veterinary_record(2020,"All good")

In [48]: # View Fidos records
         dog1.veterinary_record

Out[48]: [(2020, 'All good')]

In [50]: # Fluffy has no records
         dog2.veterinary_record

Out[50]: []
```

Figure 22: Example Python Class

The `__init__` method seen in Figure 22 is a required method in the class and is called a constructor in OOP. It is an initialization method that is run every time a new instance of the class is made and is used to set the class instance's initial state. It can be passed an arbitrary quantity of parameters, but the parameter called `self` is mandatory. This is a keyword in Python to refer to the attributes of this particular instance of the class.

Inheritance: Inheritance in OOP is when a class inherits from another class, either extending it or overwriting some parts of it. In the previous example, Dog could be a sub-class of Animal, where the Animal class holds the information relevant for any animal types. The Animal class can then be sub-classed with other animal types later without rewriting the common parts. An example of this can be seen in Figure 23.



```
In [62]: class Animal:
             def __init__(self, name, age):
                 self.name = name
                 self.age = age

             # Function to display information about an animal
             def display(self):
                 print("My name is "+self.name+", I am "+str(self.age)+" year old.")

         class Dog(Animal):
             def __init__(self, name, age, race):
                 # Super extends the previous __init__ function
                 super(Dog, self).__init__(name, age)
                 self.race = race
                 self.veterinary_record = []

             # Overwrites the display function
             def display(self):
                 print("My name is "+self.name+", I am a "+str(self.age)+" year old "+self.race)

             # Function to add a veterinary record
             def add_veterinary_record(self, year, record):
                 self.veterinary_record.append((year, record))

In [56]: dog = Dog("Fido", 2, "Border Collie")

In [58]: dog.display()

         My name is Fido, I am a 2 year old Border Collie

In [60]: animal = Animal("Olivia",5)

In [61]: animal.display()

         My name is Olivia, I am 5 year old.
```

Figure 23: Example of Python Inheritance

In Figure 23, the `__init__` function *extends* the same function from the parent class by using the keyword `super`, while the `display` function *overwrites* the same function from the parent class.

Loading libraries: In Python, all built-in or external libraries used in the code must be imported at the top of the Python file. It is possible to import the whole library or just specific parts of it. Examples of both can be seen in Figure 24 below.

```
# Full library import
import yaml
# Specific import
from yaml import dump
```

Figure 24: Example of importing libraries in Python

6.3 Testing during development

Testing is an essential step when writing software. It is the only way of checking that the code works as expected and needs to be done frequently during development. It is essential to find bugs, flaws, and errors as early as possible in the development process when it is still easy to fix. This type of testing is called functional testing.

It is also vital to ensure that all older programming still works after new changes are made to the code. This type of testing is called regression testing.

For the remainder of this thesis, testing will be used as a common word for both types of testing. There are primarily two ways of performing tests; manual or automatic.

6.3.1 Manual testing

Manual testing is performed simply by using the developed software. This is known as exploratory testing and is a task that the developer should often do as it is the only way of ensuring that the software is behaving as expected. It is also vital that the customer tests the software with real-life use-cases because the developer and the customer might have completely different ways of using it.

6.3.2 Automated testing

Manual testing can be very time-consuming, and we cannot expect humans to be consistent and produce the same result every time. Therefore, scripted tests are used to test the systems underlying functionality. Since it will give the same result every time, it can be used to figure out exactly which part of the system is failing.

There are generally two types of automatic tests, the unit test and the integration test (Testim, 2021).

A unit tests check that one single component of the code operates as expected. An example of a unit test can be testing that one specific function returns the expected value or performs the correct action. Unit tests are usually in the so-called white-box testing group, which means that the internal workings of the code are known to the test and test writer.

An integration test checks that different components of the system operate with each other. An example of an integration test is performing a sequence of actions in a UI to execute different system functions and check that the correct actions are taken. Integration tests can be either white-box tests or black-box tests where the internal workings of the code are unknown to the test and test writer.

Both unit tests and integration tests are utilized in this project. More details on the implementation of automated testing can be seen in chapter 7.4.

6.3.3 Test-driven development

Test-Driven Development (TDD) is an iterative process in which the developer writes the automated tests before writing the code.

A well-known TDD approach is called “Red, Green, Refactor” (Shore, 2005), described as a three-step process shown in Figure 25.

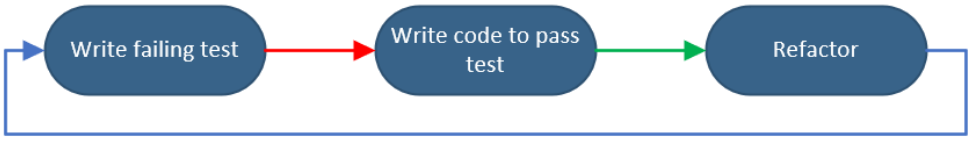


Figure 25: TDD process “Red, Green, Refactor”

- Red: Write a small failing test without thinking about how to solve it.
- Green: Write a small amount of code to satisfy the test without thinking of how the code is written. The answer could even be hardcoded if it passes the test.

- Refactor: Rewrite the code to improve it after the tests are passing. This means removing duplicated code and other faults seen in the code and rewriting the code to be more neat and tidy.

This process is repeated until the code is complete. Using this approach makes the thought process much more pleasant, as only one task has to be solved at a time. Often, this process will result in cleaner code and fewer bugs in the final product.

There are three general rules to follow when using TDD that has been proposed by (Martin, 2011):

1. It is not allowed to write any production code until first having written a failing unit test.
2. It is not allowed to write more of a unit test than is sufficient to fail.
3. It is not allowed to write more production code than is sufficient to pass the currently failing unit test.

This development approach is used for all applicable parts of the software development in this project.

6.4 Development environment and tools

This section explores the different tools and services used throughout the development process, mainly focusing on online services. The interconnection of these tools can be seen in Figure 26 below.

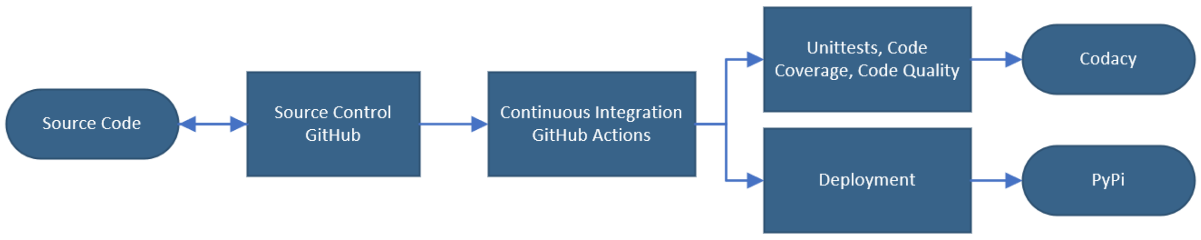

Source Code
Source Control GitHub
Continuous Integration GitHub Actions
Unittests, Code Coverage, Code Quality
Codacy
Deployment
PyPi

Figure 26: Interconnection of development tools

6.4.1 Source control and Continuous Integration

Source control is one of the essential tools in today's software development. It stores the history of all changes of the files in the project. Therefore, it can also revert to a previous version of the code if mistakes are made.

Source control is also tightly connected with Continuous Integration (CI) tools, which are used to automate tasks such as running tests, checking code quality, and deploying the code when changes are made to the source control repository.

Git (Software Freedom Conservancy, 2021) is used for source control for this project through the cloud service GitHub (GitHub Inc., 2021), which also provides the CI tool GitHub Actions. The project will live under Equinor's organization on GitHub, where it can be further developed by in-house developers later. Each user story has its own pull request on GitHub, which means that the complete history of the development of each user story can be seen in detail.

6.4.2 Test coverage

When writing and running tests on the code, it is nice to know how much of the codebase the tests cover. Tools exist that show which files and code lines are covered. An online service will be used for this and connected to the project GitHub repository to check test coverage. For this project, a service called Codacy (Codacy, 2021) is used to check test coverage.

6.4.3 Code quality

The code in this project is written according to the coding standards of the team behind Config Suite. This means following the Black code styling (Langa et al., 2020). This code style builds on a standard called PEP8 (Rossum et al., 2001). A few code checking tools will be utilized to ensure the code follows the correct styling and standards. Flake8 (Cordasco, 2016) is a tool that will check that the code follows the PEP8 standard, and the Black tool (Langa et al., 2020) will ensure that the code follows the Black code styling.

The book The Clean Coder (Martin, 2011) has also inspired how the code should be written, for example, by having self-explanatory parameter and function names and writing self-explanatory code where comments are kept at a minimum.

6.4.4 Code deployment

The project is deployed to PyPI, short for the Python Package Index (Python Software Foundation, 2021a). This service is the standard for Python package installation and ensures that everyone can install the software with a simple terminal command. The project can be found under the name “configsuite-tui” on PyPI.

6.4.5 Documentation

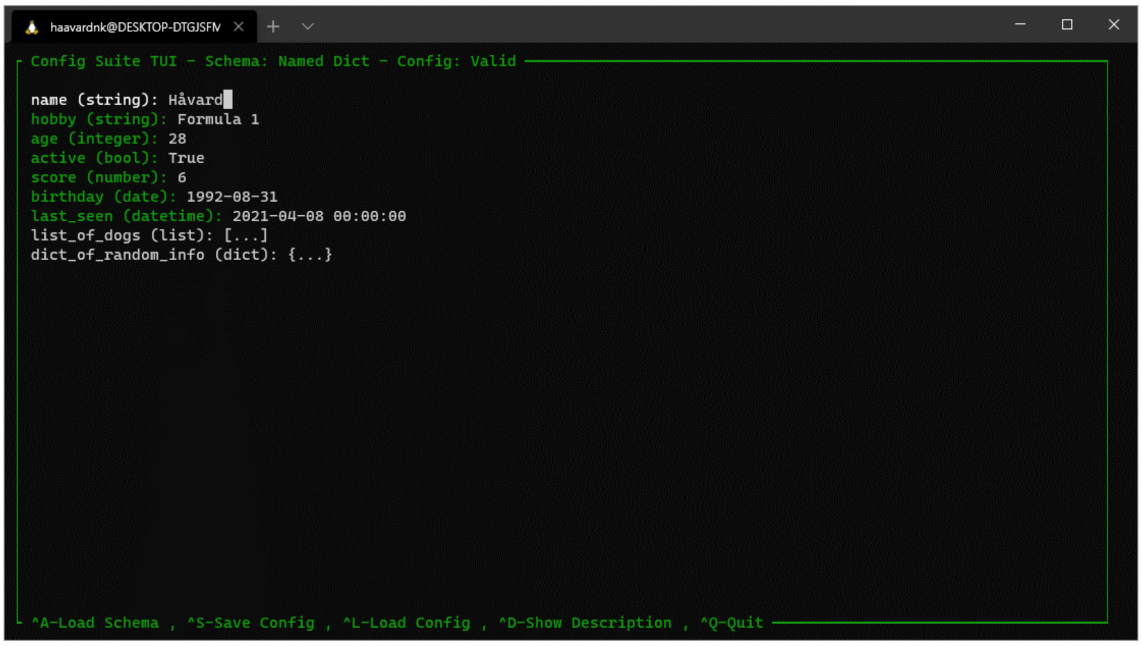
User documentation for the project is made using Sphinx (Brandl, 2021), a tool that automatically builds HyperText Markup Language (HTML) documentation from reStructuredText. The documentation is stored in the GitHub repository and is then hosted on the service Read the Docs (Read the Docs Inc., 2021). The documentation is automatically built for new commits and releases on GitHub.

6.4.6 Open-source software

Equinor is a significant contributor to open-source software. They have an open-source first strategy where the developed software should be open-source as long as there is no Intellectual Property (IP), implying that it should be closed-source. The TUI developed in this project will be made open-source under the MIT license, which lets anyone use, extend, or change the source code without restrictions.

7 Results

This chapter will present the final product developed during this project. All the major parts of the TUI will be detailed, including frontend functionality, backend functionality, testing, deployment, and examples of usage. All changes throughout the development process can be seen in the project's GitHub repository (Equinor ASA, 2021a).



```
Config Suite TUI - Schema: Named Dict - Config: Valid

name (string): Håvard
hobby (string): Formula 1
age (integer): 28
active (bool): True
score (number): 6
birthday (date): 1992-08-31
last_seen (datetime): 2021-04-08 00:00:00
list_of_dogs (list): [...]
dict_of_random_info (dict): {...}

^A-Load Schema , ^S-Save Config , ^L-Load Config , ^D-Show Description , ^Q-Quit
```

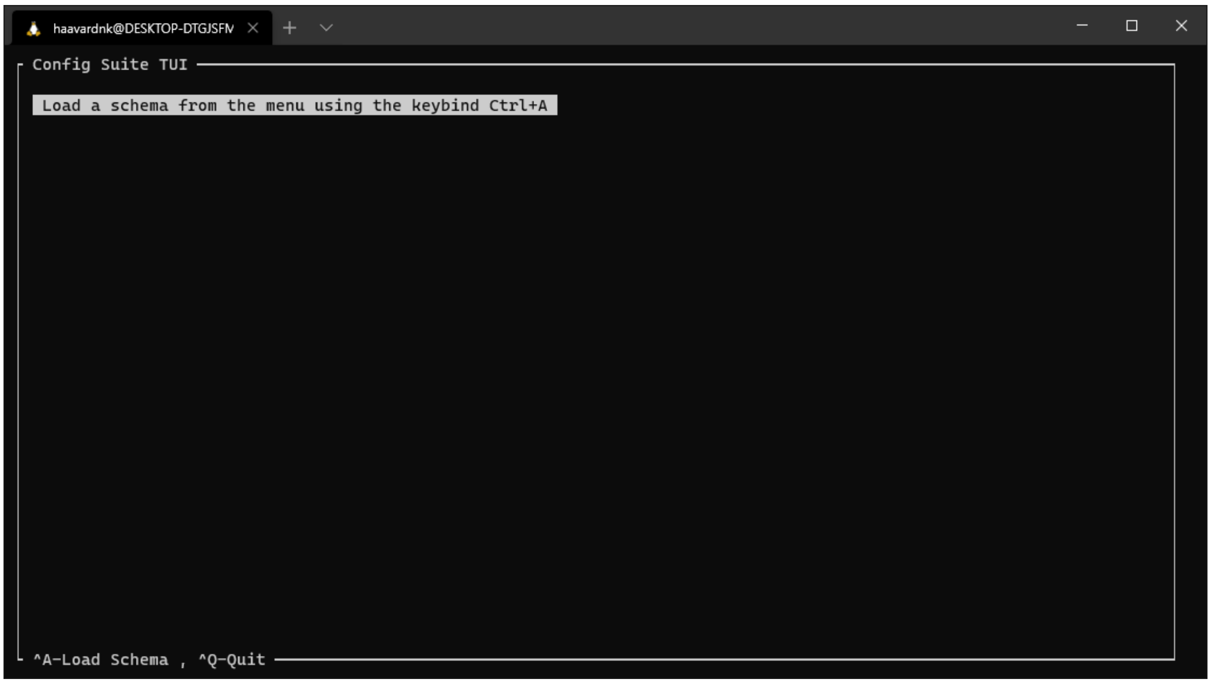
Figure 27: Screenshot of Config Suite TUI with a valid configuration

Config Suite TUI was developed between February and April of 2021 as the central part of this thesis. It is a Python-based UI extension for Config Suite, which runs in the terminal on Linux-based systems. A screenshot of the UI can be seen in Figure 27. More details about Config Suite can be found in chapter 3.

The TUI was developed over four Agile iterations, which included four epics and 23 user stories.

7.1 Usage

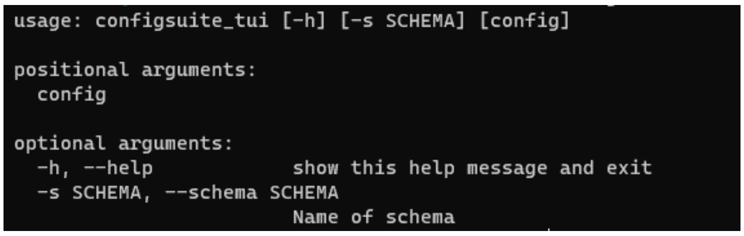
Config Suite TUI is installed with the command `pip install configsuite-tui`, after which it can be opened by typing `configsuite_tui` in the terminal. Launching the application this way presents the user with the screen seen in Figure 28, which asks the user to load one of the available schemas.



```
Config Suite TUI

 Load a schema from the menu using the keybind Ctrl+A

^A-Load Schema , ^Q-Quit
```

Figure 28: UI after launching without options

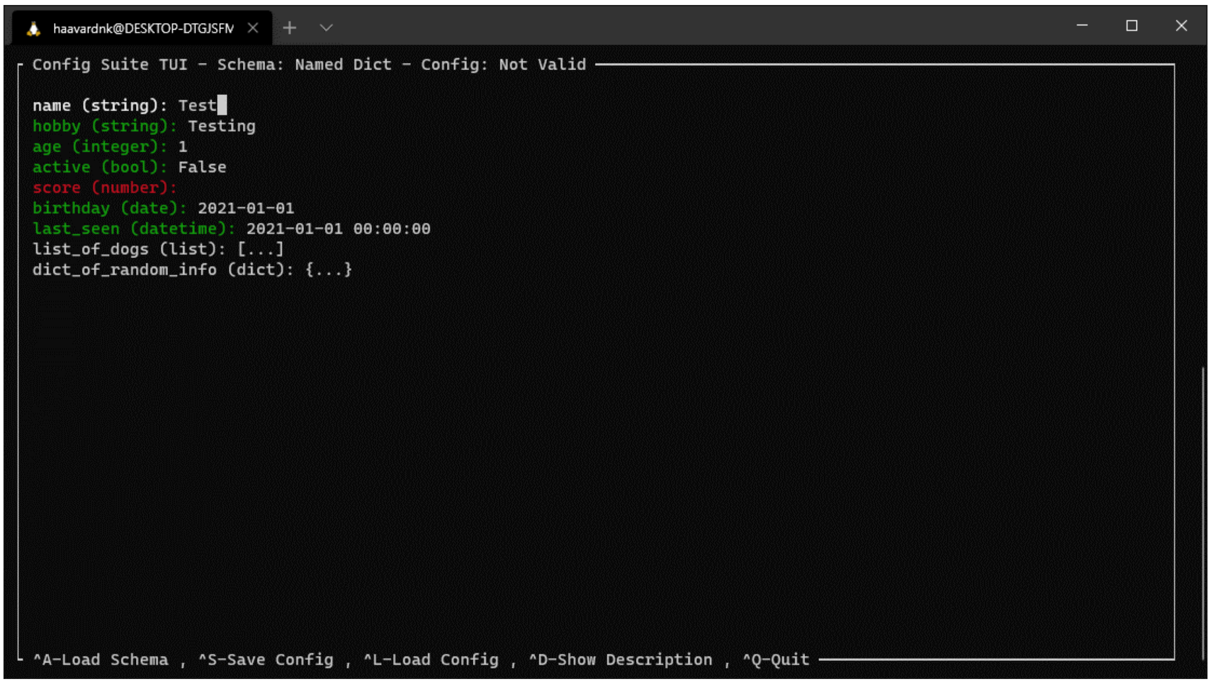
The TUI can also be launched with command-line arguments to load a schema or configuration at execution time. Configurations can also be passed without a schema if the configuration file has the schema name annotated at the top, as shown in chapter 7.3.1 below. The launch arguments can be seen in Figure 29.



```
usage: configsuite_tui [-h] [-s SCHEMA] [config]

positional arguments:
  config

optional arguments:
  -h, --help            show this help message and exit
  -s SCHEMA, --schema SCHEMA
                        Name of schema
```

Figure 29: Command-line arguments for the TUI

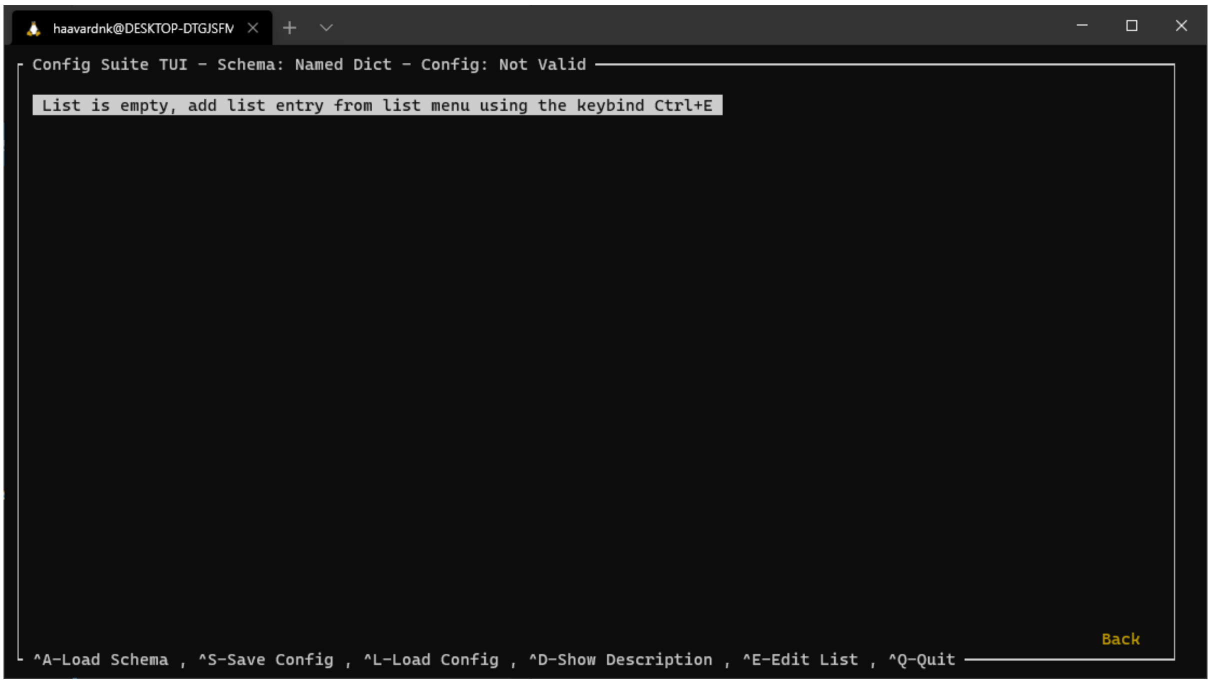
If the passed arguments are valid and both the schema and configuration exist, the TUI will launch with these already loaded and immediately render them.



```
haavardnk@DESKTOP-DTGJSFM
Config Suite TUI - Schema: Named Dict - Config: Not Valid

name (string): Test
hobby (string): Testing
age (integer): 1
active (bool): False
score (number):
birthday (date): 2021-01-01
last_seen (datetime): 2021-01-01 00:00:00
list_of_dogs (list): [...]
dict_of_random_info (dict): {...}

^A-Load Schema , ^S-Save Config , ^L-Load Config , ^D-Show Description , ^Q-Quit
```

Figure 30: UI after loading schema

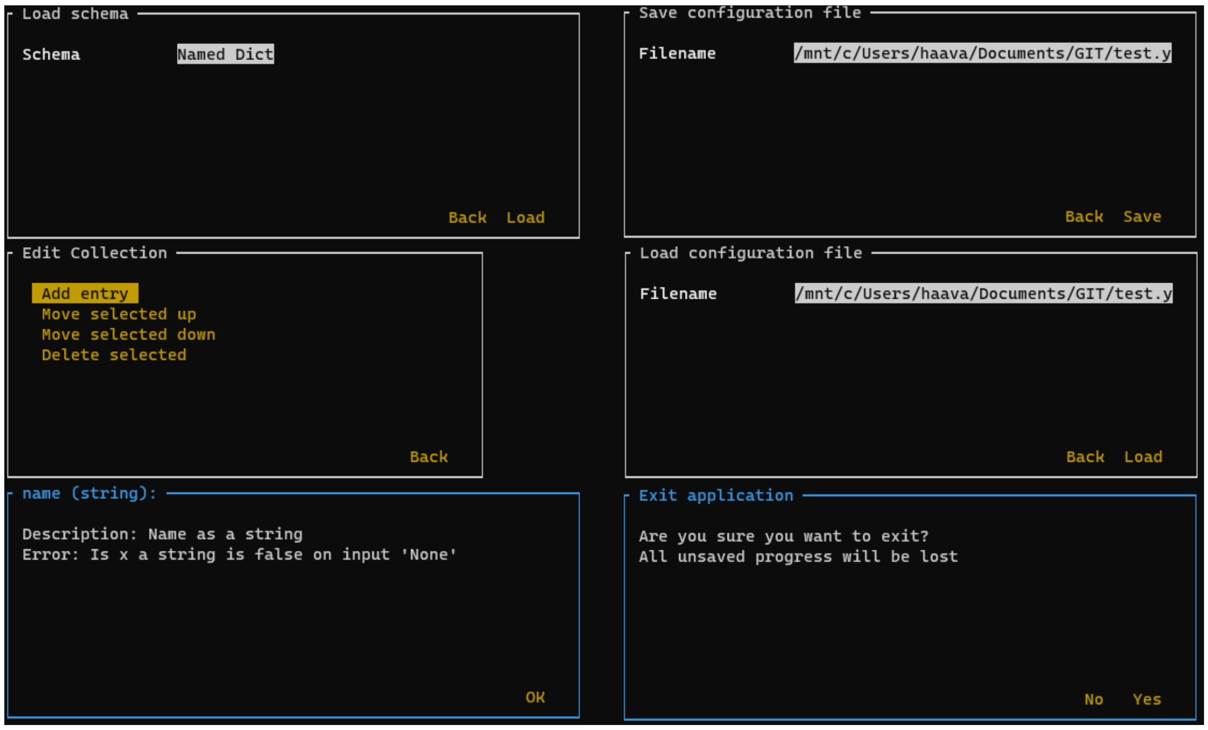
Figure 30 shows an example where both a schema and configuration are loaded. Now the TUI is completely initiated and shows all applicable menu shortcuts for this view. This schema has both a List and a Dictionary inside it. These can be opened by clicking on them and will open as shown in Figure 31.



```
haavardnk@DESKTOP-DTGJSFM
Config Suite TUI - Schema: Named Dict - Config: Not Valid

 List is empty, add list entry from list menu using the keybind Ctrl+E

                                                                          Back
^A-Load Schema , ^S-Save Config , ^L-Load Config , ^D-Show Description , ^E-Edit List , ^Q-Quit
```

Figure 31: UI after opening a dictionary

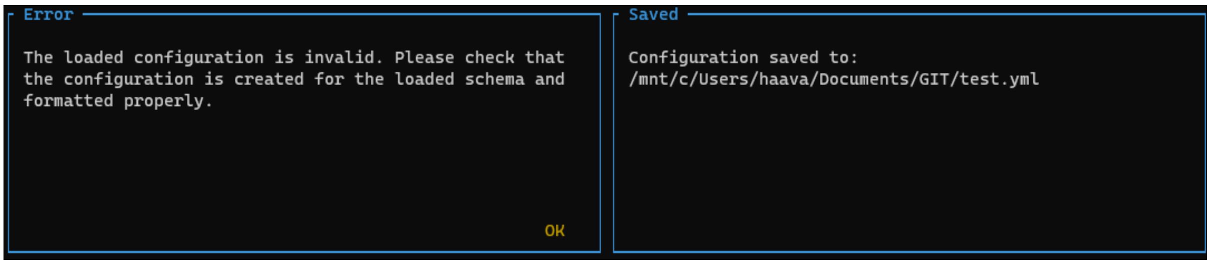
All shortcuts in the application are shown in Figure 31. A collage showing all the different menus of the TUI can be seen in Figure 32.



```
Load schema
Schema          Named Dict
                                                Back  Load

Edit Collection
  Add entry
  Move selected up
  Move selected down
  Delete selected
                                        Back

name (string):
Description: Name as a string
Error: Is x a string is false on input 'None'
                                                      OK

Save configuration file
Filename        /mnt/c/Users/haava/Documents/GIT/test.y
                                                Back  Save

Load configuration file
Filename        /mnt/c/Users/haava/Documents/GIT/test.y
                                                Back  Load

Exit application
Are you sure you want to exit?
All unsaved progress will be lost
                                                  No   Yes
```

Figure 32: All menus of the TUI

In addition to the menus, there are popup messages to inform the users of errors and events. Examples of an error and an informational popup can be seen in Figure 33.



```
Error
The loaded configuration is invalid. Please check that
the configuration is created for the loaded schema and
formatted properly.
                                                      OK

Saved
Configuration saved to:
/mnt/c/Users/haava/Documents/GIT/test.yml
```

Figure 33: Examples of popup messages from the TUI

7.2 Frontend

The frontend of the application is the part that is visible to the user. The focus here is to deliver a good user experience and present the information generated by the application's backend logic. This subchapter will detail all the different sub-parts of the frontend functionality displayed in the previous subchapter.

7.2.1 Frontend Library

The UI is built with a community library called npyscreen (Cole, 2014), a TUI library for Python. It is based on Curses, "a library that supplies a terminal-independent screen-painting and keyboard-handling facility for text-based terminals" (Kuchling & Raymond, 2021). This is initially a low-level C library that was written for BSD Unix, but a Python wrapper for Curses is built into Python.

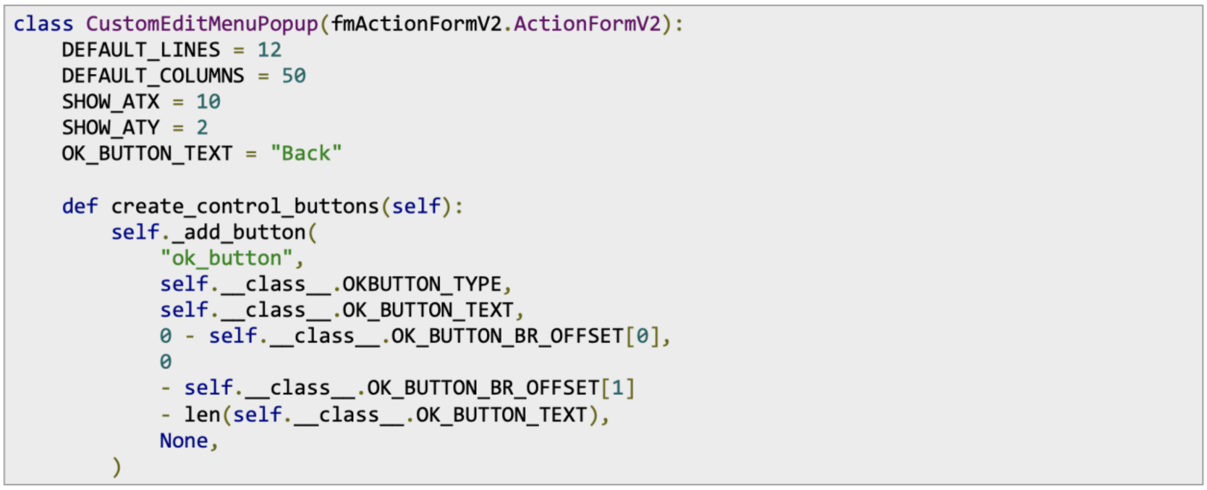
The TUI library provides a more high-level approach than using Curses directly by providing widgets and premade objects, making the developer's job much more manageable.

This library was chosen over seven other TUI libraries. The reasons for choosing npyscreen are simple; it was the only library that provided a good testing system and had great documentation.

7.2.2 Custom components

The TUI library npyscreen comes with a selection of built-in widgets for forms, popups, and menus. Common for them is that they are general and do not always fit perfectly with the wanted implementation.

All the components used in this project have been customized to some extent. Examples of this are all button texts in menus and popups, custom footers on the forms, and custom functions for handling UI state. These customizations are made by using inheritance and superclasses to overwrite and extend the widgets. More about inheritance and superclasses can be read about in chapter 6.2. An example of this can be seen in Figure 34 below.



```python
class CustomEditMenuPopup(fmActionFormV2.ActionFormV2):
    DEFAULT_LINES = 12
    DEFAULT_COLUMNS = 50
    SHOW_ATX = 10
    SHOW_ATY = 2
    OK_BUTTON_TEXT = "Back"

    def create_control_buttons(self):
        self._add_button(
            "ok_button",
            self.__class__.OKBUTTON_TYPE,
            self.__class__.OK_BUTTON_TEXT,
            0 - self.__class__.OK_BUTTON_BR_OFFSET[0],
            0
            - self.__class__.OK_BUTTON_BR_OFFSET[1]
            - len(self.__class__.OK_BUTTON_TEXT),
            None,
        )
```

Figure 34: Custom class for Edit Collection menu

The code snippet in the example is the custom popup class for the "Edit Collection" menu, as seen in the second row to the left in Figure 32. The first five lines overwrite the window size, placement, and button text. The function overwrites the button creation because the original widget had two buttons, but only one was needed.

7.2.3 Rendering Config Suite schema

After a schema is loaded and processed by the application backend, it is rendered to the screen as widgets. There is one widget for each field in the schema, where the widget variant and settings are type-dependent.

There are four different ways the TUI renders the fields. Integer, String, Float, Date, and DateTime are all rendered as a text input field but are processed differently in the backend to convert to the right type and validate the input.

Booleans are rendered as a multiple-choice field, where the only possible choices are true or false. Lists and Dictionaries are rendered as static text buttons, which opens a new page with the respectable List or Dictionary. Examples of all these can be seen in Figure 30 above.

7.2.4 Frontend to backend

The frontend and backend of the application are very tightly connected. All backend actions are executed based on actions done by the user. Some vital backend functions are executed often. For example, the function that updates and validates the configuration and updates the application state is executed every time the user pushes a button on their keyboard. This means that the configuration gets updated and validated for every character written in a field and lets the user feel like the application is fast and works in real-time.

7.3 Backend

The backend of the application is like the engine of a car. It lives under the hood, but it is what drives the application. This subchapter will detail all the different sub-parts of the backend functionality of the TUI.

7.3.1 Working with YAML files

One of the primary tasks of the application is working with configuration files. An external community library called PyYAML (Simonov, 2016) is used for reading and writing YAML formatted files. When writing a file and saving the configuration, the TUI also annotates the currently used schema into the file header. This makes it possible for the TUI to load a configuration without specifying a schema. The header format can be seen in Figure 35 below.

```
!# configsuite-tui: schema=owned_cars
```

Figure 35: Header format of the annotated configuration file

Figure 36 below shows the functions for working with the YAML files.

```python
def load(filename):
    with open(filename, "r") as file:
        first_line = file.readline()

    with open(filename) as file:
        schema = None
        if "!# configsuite-tui" in first_line:
            next(file)
            schema = first_line.split("!# configsuite-tui: schema=")[1].strip()
        return yaml.full_load(file), schema

def save(config, filename, schema):
    with open(filename, "w") as file:
        file.write("!# configsuite-tui: schema=" + schema + "\n")
        yaml.dump(config, file, sort_keys=False)
```

Figure 36: Functions for working with YAML files

These functions read and write the configuration between a Python Dict/List and the YAML file. The function first extracts or writes the header before loading or dumping the YAML information. Processing of loaded configurations can be seen in chapter 7.3.5 below.

7.3.2 Schema Plugin System

Config Suite schemas are supplied to the TUI through a plugin system. This plugin system makes it possible for other Python modules to register their schemas with the TUI on installation. This way, all installed Config Suite schemas from external packages become automatically available in the TUI.

The plugin system is made using the community library Pluggy (Krekel, 2016), which implements a hook system. The hook system is split into two parts; the hook specification and the hook implementation. The host application (the TUI) provides the hook specification, an empty function that other applications can fill for use in the host application.

The hook specification for the TUI is simple and can be seen in Figure 37 below.

```
import pluggy

hookspec = pluggy.HookspecMarker("configsuite_tui")

@hookspec
def configsuite_tui_schema():
    """Register Config Suite schema in Config Suite TUI
    Return dict: {name: schema}
    name = string
    schema = Config Suite schema object
    """
```

Figure 37: Schema plugin hook specification

The hook implementation is a program-specific implementation of the hook specification. This is implemented in every Python package which wants to register a schema with the TUI. An example of a package using this hook specification to register its schema can be seen in Figure 38 below.

```
import configsuite_tui
from configsuite import types
from configsuite import MetaKeys as MK

@configsuite_tui.hookimpl
def configsuite_tui_schema():
    name = "Plugin Test"
    schema = {
        MK.Type: types.NamedDict,
        MK.Content: {
            "name": {MK.Type: types.String},
            "hobby": {MK.Type: types.String},
            "age": {MK.Type: types.Integer},
        },
    }

    return {name: schema}
```

Figure 38: Example of schema plugin hook implementation

The hook implementation must then be registered by linking the module to the host application through Setuptools (Python Packaging Authority, 2021), the framework used to install Python libraries and packages on a computer. Figure 39 below shows how the example hook implementation is registered in the system.

```python
from setuptools import setup

setup(
    name="configsuite-tui-plugin",
    install_requires="configsuite_tui",
    entry_points={"configsuite_tui": ["plugin = configsuite_tui_plugin"]},
    py_modules=["configsuite_tui_plugin"],
)
```

Figure 39: Registering the plugin with Setuptools

7.3.3 Interaction with Config Suite

Config Suite can be thought of as the parent package to the TUI. This means that a lot of the functionality of the application comes directly from communicating with Config Suite. More about Config Suite can be seen in chapter 3.

There are two helper functions made to take care of Config Suite functionality. One which checks whether the configuration is valid and collects all eventual errors. The other function checks whether the configuration is readable. They can be seen in Figure 40 below.

```python
def validate(config, schema, index):
    suite = ConfigSuite(config, schema, deduce_required=True)
    error_list = get_page_errors(suite.errors, index)
    return suite.valid, error_list

def get_page_errors(errors, index):
    # Returns only the errors belonging to the current page
    error_list = {}
    for error in errors:
        if error.key_path[0:-1] == tuple(index):
            error_list[error.key_path[-1]] = error.msg
    return error_list

def readable(config, schema):
    return ConfigSuite(config, schema, deduce_required=True).readable
```

Figure 40: Config Suite related functions

The validation function is executed every time a change is made in the TUI. This provides instant feedback to the user on whether the inputted value is valid or not. The readable function is executed when loading a configuration file to assess if the configuration can be loaded with the given schema or not.

7.3.4 Config Suite Schema

A schema must be loaded for the TUI to work. When loaded, the schema is stored as its original Config Suite configuration object in the application.

The TUI supports all standard functionality in Config Suite, meaning it supports all the standard types and collections listed in chapter 3. It will also support all custom types that can be rendered as a string field, but there is no guarantee that validation will work as expected. It can, however, easily be extended to support any custom type completely.

The TUI will also extract optional description texts and eventual error messages from the schema to present to the users.

7.3.5 Configuration processing

Around 70% of the application code is for processing and manipulating the entered configuration based on frontend actions. The application uses many parameters for tracking the application's states, including the entered configuration, selected schema, current indexes of the application, errors, and validation information.

The configuration state parameter is a dynamically generated data structure with nested Lists and Dictionaries that store all information entered into the TUI. Since some of the collections in a Config Suite schema have an arbitrary length, the state parameter structure cannot be entirely generated upon creation. This is challenging to solve because of how data structures work in Python.

Python lists are easy to work with as they can be extended, reduced, and changed. Dictionaries are worse since keys must be removed and re-added to rename the key, and new keys will always be added to the end of the Dictionary. Therefore the Dictionaries must be re-created to change a key and still have it in the same position.

This could be solved using an OrderedDict in Python instead of the ordinary Dictionary, but it was solved by creating another temporary configuration state parameter, which only holds the currently selected page's structure information. This makes it easy to recreate and reshape the structure and then pass it back to the main state parameter.

This again proposes a new challenge: returning the newly reshaped structure into the correct location in the nested structure of the main state parameter. Because of this, the application only supports a set number of nested levels. For this project, it was agreed with the customer to include four levels, including the base level, which can easily be extended in the future. A third state parameter holds the indexing information needed to return the position-based configuration to its correct location in the main state parameter.

The configuration in Figure 41 below will be used as an example to show how the configuration processing works.

```
config = {
    "name": "Håvard",
    "owned_cars": [
        {
            "car_type": "Porsche 911",
            "previous_owners": [
                "Thomas",
                "Bernt"
            ]
        },
        {
            "car_type": "BMW M3",
            "previous_owners": [
                "Erik",
                "Kjell"
            ]
        },
    ],
}
```

Figure 41: Example four-level configuration

This configuration is a four-level nested schema of a person’s owned cars and their previous owners.

Imagine, using the given configuration above, that the user is currently editing the `previous_owners` List of the first car. The temporary state parameter would then look seen in Figure 42 below.

```
page_config = ["Thomas","Bernt"]
```

Figure 42: Position-based state parameter after selecting previous_owners of the first car

The index parameter would, in this case, hold the information seen in Figure 43 below.

```
index = ["owned_cars", 0, "previous_owners"]
```

Figure 43: Index parameter after selecting previous_owners of the first car

After the user has changed the currently selected configuration, i is passed back to the main state parameter. Python cannot dynamically index a Dictionary or List when we want to assign a new value to it; therefore, each index level must be done explicitly, resulting in the TUI only supporting a chosen number of nested collections as described above. The solution is to assign the values back to the main state parameter based on the length of the index parameter, as seen in Figure 44 below,

```
if len(index) == 1:
        config[index[0]] = page_config
elif len(index) == 2:
        config[index[0]] [index[1]] = page_config
elif len(index) == 3:
        config[index[0]] [index[1]] [index[2]] = page_config
```

Figure 44: Functions to return the selected configuration back into the main configuration

The result of this is a well-functioning workaround for how Python handles collections and gaining more control of the state of the application by only examining parts of the configuration at a time.

7.4 Automated Testing

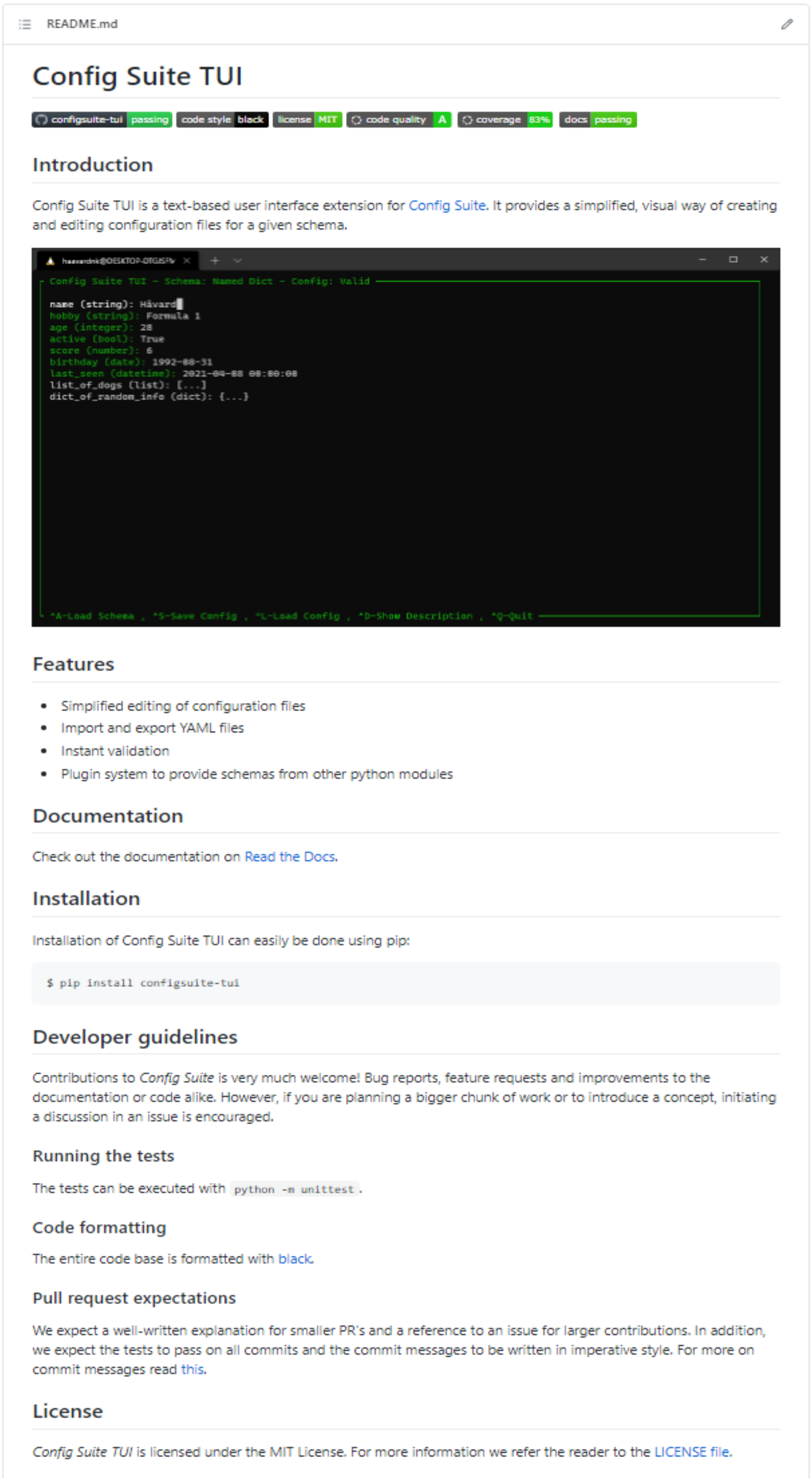
Automated testing is done using the built-in Python library Unittest (Python Software Foundation, 2021c). Due to the nature of the system, unit tests are written for the backend functions working with YAML files and Config Suite, and integration tests are written for testing the TUI.

Running tests for a visual application is tricky because of the need to simulate user actions in the UI and check that the correct response is given both for the application's frontend and backend. This functionality is not built into most Python test runners, including Unittest.

Npyscreen comes with a built-in test functionality where it is possible to pass a sequence of buttons to push when launching the test. All the UI tests are made so that the test runner launches the application and clicks through the button sequence before exiting the application. The response can then be asserted against expected values. An example of an integration test can be seen in Appendix A. That is one of the primary integration tests which tests loading a schema, filling it, and saving the file before asserting the output from the TUI and the contents of the saved file.

The tests are executed automatically every time a change in the code is pushed to the GitHub repository and will block code merging if the test fails. This proved to be a challenge, as testing a TUI requires a screen output, and the test runner on GitHub's CI tool does not provide one. This issue was overcome by adding a community library called Invoke (Forcier, 2020). This library makes it possible to run commands, including Unittest, inside an emulated terminal, also called a pseudo-terminal, with simulated screen output.

7.5 Deployment



README.md

Config Suite TUI

configsuite-tui passing | code style black | license MIT | code quality A | coverage 83% | docs passing

Introduction

Config Suite TUI is a text-based user interface extension for Config Suite. It provides a simplified, visual way of creating and editing configuration files for a given schema.

```
haavardnk@DESKTOP-DTGJSFW
┌ Config Suite TUI - Schema: Named Dict - Config: Valid ─────────────────

  name (string): Håvard█
  hobby (string): Formula 1
  age (integer): 28
  active (bool): True
  score (number): 6
  birthday (date): 1992-08-31
  last_seen (datetime): 2021-04-08 08:00:08
  list_of_dogs (list): [...]
  dict_of_random_info (dict): {...}

└ ^A-Load Schema , ^S-Save Config , ^L-Load Config , ^D-Show Description , ^Q-Quit ─────
```

Features

- Simplified editing of configuration files
- Import and export YAML files
- Instant validation
- Plugin system to provide schemas from other python modules

Documentation

Check out the documentation on Read the Docs.

Installation

Installation of Config Suite TUI can easily be done using pip:

```
$ pip install configsuite-tui
```

Developer guidelines

Contributions to *Config Suite* is very much welcome! Bug reports, feature requests and improvements to the documentation or code alike. However, if you are planning a bigger chunk of work or to introduce a concept, initiating a discussion in an issue is encouraged.

Running the tests

The tests can be executed with `python -m unittest`.

Code formatting

The entire code base is formatted with black.

Pull request expectations

We expect a well-written explanation for smaller PR's and a reference to an issue for larger contributions. In addition, we expect the tests to pass on all commits and the commit messages to be written in imperative style. For more on commit messages read this.

License

Config Suite TUI is licensed under the MIT License. For more information we refer the reader to the LICENSE file.

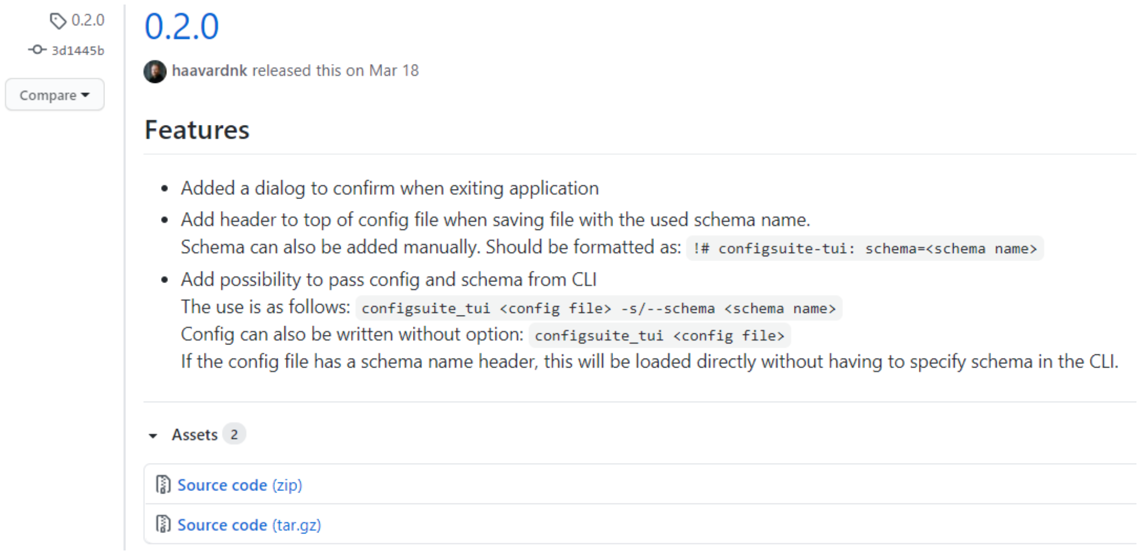
Figure 45: Readme from Config Suite TUI GitHub Repository (Equinor ASA, 2021a)

GitHub is the most central location for the whole project. Here the source code is hosted, and GitHub Actions execute CI tasks, including automated testing and deployment. The screenshot in Figure 45 shows the readme which users are faced with on GitHub.

The software is licensed under the MIT license on GitHub. The license file in the base folder explains the content of the license and Equinor's copyright.

7.5.1 Versioning and releases

Official releases of the application are created manually on GitHub. These are created from a chosen Git commit and include release notes and source code for the actual release. The releases are tagged with a version number, which follows semantic versioning (Preston-Werner, 2021). An example of a release with version 0.2.0 can be seen in Figure 46.



0.2.0

haavardnk released this on Mar 18

Features

- Added a dialog to confirm when exiting application
- Add header to top of config file when saving file with the used schema name.
  Schema can also be added manually. Should be formatted as: `!# configsuite-tui: schema=<schema name>`
- Add possibility to pass config and schema from CLI
  The use is as follows: `configsuite_tui <config file> -s/--schema <schema name>`
  Config can also be written without option: `configsuite_tui <config file>`
  If the config file has a schema name header, this will be loaded directly without having to specify schema in the CLI.

Assets 2

Source code (zip)

Source code (tar.gz)

Figure 46: Release notes for version 0.2.0

Config Suite TUI has had eleven releases so far, where the final delivery of this project is at version 0.2.3.

All releases are automatically uploaded using GitHub Actions to Equinor's PyPi repository with the assigned version and source code. Using PyPi, users can easily install Config Suite TUI using the terminal command `pip install configsuite-tui` as long as they have Python version 3.6 or higher installed on their computer.

8 Discussion

This project has attempted to improve an already great Python configuration library called Config Suite by extending it with an Intelligent User Interface (IUI). There are many ways to create an IUI, but we came up with four possible concepts by brainstorming together with the customer.

The concepts were further matured before making a final selection. The chosen concept was to make a Text-based User Interface (TUI), chosen over a Graphical User Interface (GUI) and a Visual Studio Code extension. The reasons for choosing a classical TUI were based on a technical evaluation and the customer's desires. After finishing the development, it still looks like a solid choice, with high performance, an intuitive plugin system, and a clean user interface.

By choosing this concept, the development language was already set in stone. Since Config Suite is written in Python, it would not be wise to make the TUI in any other language. When using Python, many official and community libraries are used for different functions. For the UI library itself, eight different libraries were examined in detail before selecting Npyscreen. This library has given many challenges that had to be worked around by creating custom components for most functions of the TUI. It has worked fine overall and provided a solid testing functionality, which was one of the main reasons for choosing it.

The TUI was developed using Agile methodologies, and it was attempted to replicate how a developer team in Equinor works as much as possible for a one-person team. The primary parts adopted from Agile were the usage of user stories, epics, and continuous user feedback. This has been a great contributor to organizing and developing the TUI in a controlled manner. The other adopted tools, such as Kanban from Agile and Gantt from Lean, were used less than planned since both are organization tools for teams, and since a one-person team develops this project, it was more of an extra job to keep them updated. One of the 12 principles of the Agile manifesto recommends reflecting on how to become more effective at regular intervals, and if this software development were to continue any longer, these two tools would have been discarded.

The final product is a functional UI that fulfills the goal of making it easier for users to fill in configuration files without knowing anything about the context of the application. It does, however, require the developers to enable the TUI by adding a hook implementation into their application, so the schema(s) of their application becomes visible to the TUI's plugin system.

The goal was to make a UI with intelligent features. The delivered TUI does not have intelligence in the form of machine learning or artificial intelligence, but it makes the users' life easier by adapting and automating many of the sub-tasks of creating and editing a configuration for Config Suite, which is some of the most used terms for describing intelligent systems.

Modern software development tools were used during the development process, and the TUI extension now lives as open-source software licensed under the MIT license on Equinor's GitHub repository. Being an open-source software package means it can be used, extended, or changed by anyone in the future.

9 Conclusion

This Master's thesis project developed a Text-based User Interface (TUI) extension for Config Suite. The TUI was developed using modern software development methodologies and tools and in close collaboration with the customer, Equinor.

The resulting TUI is a solution to a challenge faced by the users of Config Suite, namely that it is challenging to work with configuration files. The TUI solves this by providing an easy-to-use, intelligent user interface that only lets users do actions valid for the chosen schema, instantly validate their input, and auto-generate the configuration file in the correct format. This means that the users no longer have to think about correct YAML formatting or invalid input.

Because of the plugin system of the TUI, the users do not have to think about adding the schemas they need to the TUI since this is handled by the developers, making all installed schemas automatically available.

The application is designed to be changed and extended by other developers to support their needs. This is done by having a high coding standard and using known open-source development tools. It also has a high degree of test coverage, which is unusual for similar open-source TUIs, where usually only the backend functions are tested and not the TUI itself. This will help developers by allowing them to extend the application without breaking the current functionality. It is also supplied with user documentation, developer documentation in the readme, and automatic code checking and deployment to PyPi.

The delivered application is considered complete and "free of bugs," meaning all user stories are implemented, and all discovered bugs up to the date of this thesis have been fixed. Therefore, it is ready to be enjoyed by the users of Config Suite and will hopefully be a valuable tool for software configuration in Equinor.

References

Adamopoulou, E., & Moussiades, L. (2020). An Overview of Chatbot Technology. *Artificial Intelligence Applications and Innovations: 16th IFIP WG 12.5 International Conference, AIAI 2020, Neos Marmaras, Greece, June 5–7, 2020, Proceedings, Part II, 584*, 373-383. https://doi.org/10.1007/978-3-030-49186-4_31

Agile Thought. (2021). Getting Started with Kanban. Retrieved 2021, from https://agilethought.com/blogs/getting-started-with-kanban/

Atlassian. (2021). Agile project management - User Stories. Retrieved 2021, from https://www.atlassian.com/agile/project-management/user-stories

Bailey, C. (2020). Dynamic vs Static. https://realpython.com/lessons/dynamic-vs-static/

Beck, K., Beedle, M., Bennekum, A. v., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., & Thomas, D. (2001). *Manifesto for Agile Software Development*. http://agilemanifesto.org/

Bigelow, S. J. (2018). Definition: Configuration File. https://searchitoperations.techtarget.com/definition/configuration-file

Brandl, G. (2021). *Sphinx*. [Computer Software]. https://www.sphinx-doc.org/en/master/

Chen, J. W., & Zhang, J. (2007, Oct 11). Comparing Text-based and Graphic User Interfaces for novice and expert users. *AMIA Annu Symp Proc, 2007*, 125-129.

Codacy. (2021). *Codacy*. [Computer Software]. https://www.codacy.com/

Cordasco, I. S. (2016). *Flake8*. [Computer Software]. https://pypi.org/project/flake8/

Eastwood, B. (2020). The 10 Most Popular Programming Languages to Learn in 2021. https://www.northeastern.edu/graduate/blog/most-popular-programming-languages/

Equinor ASA. (2018a). *Config Suite*. [Computer Software]. https://github.com/equinor/configsuite

Equinor ASA. (2018b). Config Suite Documentation. https://configsuite.readthedocs.io/en/stable/index.html

Equinor ASA. (2021a). *Config Suite TUI.* [Computer Software]. https://github.com/equinor/configsuite-tui

Equinor ASA. (2021b). *Diggy.* [Computer Software].

Forcier, J. (2020). *Pyinvoke.* [Computer Software]. https://www.pyinvoke.org/

GitHub Inc. (2021). *GitHub.* [Computer Software]. https://github.com/

Goodger, D. (2021). reStructuredText. https://docutils.sourceforge.io/docs/ref/rst/restructuredtext.html

Kanbanize. (2021). What is kanban? https://kanbanize.com/kanban-resources/getting-started/what-is-kanban

King, A. (2019). Parse, don’t validate. https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/

Krasnoselsky, V. (2018). *TelegramTUI.* [Computer Software]. https://github.com/vtr0n/TelegramTUI

Krekel, H. (2016). *Pluggy.* [Computer Software]. https://github.com/pytest-dev/pluggy

Kuchling, A. M., & Raymond, E. S. (2021). Curses Programming with Python. Retrieved 02.02.2021, from https://docs.python.org/3/howto/curses.html

Langa, Ł., Willing, C., Meyer, C., Zijlstra, J., Naylor, M., Dollenstein, Z., & Lees, C. (2020). *Black.* [Computer Software]. https://github.com/psf/black

Martin, R. C. (2011). *The Clean Coder: A Code of Conduct for Professional Programmers* (1 ed.). Prentice Hall.

Microsoft. (1997). *Microsoft Office Assistant.* https://en.wikipedia.org/wiki/Office_Assistant

Microsoft. (2021). *Editing JSON with Visual Studio Code.* [Computer Software]. https://code.visualstudio.com/docs/languages/json

Patton, J. (2020). Kanban Development Oversimplified. Retrieved 2021, from https://www.jpattonassociates.com/kanban_oversimplified/

Preston-Werner, T. (2021). Semantic Versioning. https://semver.org/

Python Packaging Authority. (2021). *Setuptools.* [Computer Software]. https://pypi.org/project/setuptools/

Python Software Foundation. (2021a). *The Python Package Index.* [Computer Software]. https://pypi.org/

Python Software Foundation. (2021b). *Tkinter.* [Ccomputer Software]. https://docs.python.org/3/library/tkinter.html

Python Software Foundation. (2021c). *Unittest.* [Computer Software]. https://docs.python.org/3/library/unittest.html

Read the Docs Inc. (2021). *Read the Docs.* [Computer Software]. https://readthedocs.org/

Rossum, G. v., Warsaw, B., & Coghlan, N. (2001). PEP 8 -- Style Guide for Python Code. https://www.python.org/dev/peps/pep-0008/

Simonov, K. (2016). *PyYAML.* [Computer Software]. https://pyyaml.org/wiki/PyYAML

Software Freedom Conservancy. (2021). *Git.* [Computer Software]. https://git-scm.com/about/trademark

Testim. (2021). Unit Test vs. Integration Test: Tell Them Apart and Use Both. https://www.testim.io/blog/unit-test-vs-integration-test/

TutorialsTeacher. (2020). UI in Python-Tkinter. https://www.tutorialsteacher.com/python/create-gui-using-tkinter-python

Völkel, S. T., Schneegass, C., Eiband, M., & Buschek, D. (2020). What is "Intelligent" in Intelligent User Interfaces? A Meta-Analysis of 25 Years of IUI. *arXiv*. https://arxiv.org/pdf/2003.03158.pdf

Wikipedia contributors. (2021). Python (programming language). Retrieved 14 March 2021, from https://en.wikipedia.org/w/index.php?title=Python_(programming_language)&oldid=1011721004

Appendices

Appendix A: Unittest example

```python
class Test_Tui_With_Files(TestCase):
    pm = pluggy.PluginManager("configsuite_tui")
    pm.add_hookspecs(hookspecs)
    pm.load_setuptools_entrypoints("configsuite_tui")
    pm.register(test_schema_1)

    def setUp(self):
        # Create temporary directory
        self.tmpdir = tempfile.mkdtemp()
        os.chdir(self.tmpdir)

    def tearDown(self):
        shutil.rmtree(self.tmpdir)

    @mock.patch("configsuite_tui.tui.get_plugin_manager", return_value=pm)
    def test_tui_input_save_return_validate(self, mocked_pm):
        with tempfile.NamedTemporaryFile(dir=self.tmpdir) as tmpfile:
            testinput = [
                curses.KEY_DOWN,
                "Jane Doe",
                curses.KEY_DOWN,
                "Electrician",
                curses.KEY_DOWN,
                "35",
                curses.KEY_DOWN,
                curses.ascii.NL,
                curses.KEY_DOWN,
                curses.ascii.NL,
                curses.KEY_DOWN,
                "45.35",
                curses.KEY_DOWN,
                "1999-01-01",
                curses.KEY_DOWN,
                "2011-11-04T00:05:23",
                curses.KEY_UP,
                curses.KEY_UP,
                curses.KEY_UP,
                curses.KEY_UP,
                curses.KEY_UP,
                curses.KEY_UP,
                curses.KEY_UP,
                "^S",
                curses.ascii.NL,
                curses.KEY_RIGHT,
                curses.ascii.NL,
                curses.ascii.NL,
                curses.KEY_DOWN,
                curses.KEY_DOWN,
                curses.ascii.NL,
                "^Q",
                curses.ascii.NL,
                curses.KEY_RIGHT,
                curses.ascii.NL,
            ]
            npyscreen.TEST_SETTINGS["TEST_INPUT"] = testinput
            config, valid = tui(schema="test", test=True)

            self.assertEqual(
                config,
                {
```

```
                "name": "Jane Doe",
                "hobby": "Electrician",
                "age": 35,
                "active": True,
                "score": 45.35,
                "birthday": datetime.date(1999, 1, 1),
                "last_seen": datetime.datetime(2011, 11, 4, 0, 5, 23),
            },
        )
        self.assertTrue(valid)
        test_string = (
            "!# configsuite-tui: schema=test\n"
            + "name: Jane Doe\nhobby: Electrician\nage: 35\nactive: true\nscore: "
            + "45.35\nbirthday: 1999-01-01\nlast_seen: 2011-11-04 00:05:23\n"
        )
        self.assertEqual(
            tmpfile.read(),
            test_string.encode("utf_8"),
        )
```